conference
proceedings

9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)

Vancouver, BC, Canada
October 4–6, 2010

Sponsored by
USENIX
in cooperation with
ACM SIGOPS

USENIX

Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation *Vancouver, BC, Canada* *October 4–6, 2010*

ISBN 978-1-931971-79-9

USENIX Association

Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)

October 4–6, 2010
Vancouver, BC, Canada

Symposium Organizers

Program Co-Chairs

Remzi Arpaci-Dusseau, *University of Wisconsin, Madison*
Brad Chen, *Google, Inc.*

Program Committee

Dave Andersen, *Carnegie Mellon University*
Emery Berger, *University of Massachusetts Amherst*
Felipe Cabrera, *Amazon.com*
George Candea, *EPFL*
Bryan Cantrill, *Sun Microsystems, Inc.*
Pei Cao, *Google, Inc.*
Robert M. English, *Facebook, Inc.*
Bryan Ford, *Yale University*
Michael J. Freedman, *Princeton University*
Kim Hazelwood, *University of Virginia*
Jon Howell, *Microsoft Research*
Wilson Hsieh, *Google, Inc.*
Michael Isard, *Microsoft Research*
Brad Karp, *University College London*
Randy Katz, *University of California, Berkeley*
Sam King, *University of Illinois, Urbana-Champaign*
Hank Levy, *University of Washington*
Shan Lu, *University of Wisconsin, Madison*
Ed Nightingale, *Microsoft Research*
Christopher Olston, *Yahoo! Research*
Adrian Perrig, *Carnegie Mellon University*
Vijayan Prabhakaran, *Microsoft Research*
Mendel Rosenblum, *Stanford University*
Jiri Schindler, *NetApp, Inc.*
Bianca Schroeder, *University of Toronto*
Emin Gün Sirer, *Cornell University*
Amin Vahdat, *University of California, San Diego*
Carl Waldspurger, *VMware*
Emmett Witchel, *University of Texas, Austin*
Jay Wylie, *HP Labs*
Junfeng Yang, *Columbia University*
Nickolai Zeldovich, *Massachusetts Institute of Technology*
Lidong Zhou, *Microsoft Research*

Steering Committee

Richard Draves, *Microsoft Research*
Margo Seltzer, *Harvard School of Engineering and Applied Sciences*
Robbert van Renesse, *Cornell University*
Ellie Young, *USENIX*

Poster Session Chair

Jon Howell, *Microsoft Research*

Research Vision Session Program Committee

Sam King (Chair), *University of Illinois, Urbana-Champaign*
Shan Lu, *University of Wisconsin—Madison*
Emmett Witchel, *University of Texas, Austin*

The USENIX Association Staff

External Reviewers

Ole Agesen
William de Bruijn
Haowen Chan
Anthony Cozzie
Azadeh Farzan
Ariel Feldman
Prem Gopalan
Collin Jackson
Eyal de Lara
Wyatt Lloyd
Tim Mann
Jim Mattson
David Mazières
David Meisner
Bryan Parno
Ryan Peterson
Donald E. Porter
David Shue
Ahren Studer
Shuo Tang
Amit Vasudevan
Arun Venkataramani
Bernard Wong
Hui Xue
Ting Yang
Cristian Zamfir
Steve Zdancewic

9th USENIX Symposium on Operating Systems Design and Implementation
October 4–6, 2010
Vancouver, BC, Canada

Monday, October 4

Kernels: Past, Present, and Future

Inside the Data Center, 1

Security Technologies

Concurrency Bugs

Tuesday, October 5

Wednesday, October 6

Production Networks

Mobility

Virtualization

Message from the Program Co-Chairs

Welcome to OSDI '10, the biggest OSDI yet, with 32 papers selected from an all-time high of 199 submissions. In approaching the task of chairing OSDI, we started with the explicit intention of accepting a larger set of papers, consistent with the growth in the field. Below we outline some of the rationale behind this goal, and the process we applied to achieve it.

Computer systems research is growing as a community. We believe that progress on computer systems research is limited by manpower, not by the limits of a finite domain for interesting research. By implication, as the number of systems researchers increases, the volume of interesting research likely goes up as well. Year after year, top research programs add faculty or research positions in the systems area, while at the same time new programs establish their presence in the field, including newfound growth outside the traditionally strong geographies. The expansion of our community is consistent with the robust scientific and commercial application of computer systems research, providing a strong economic basis for this growth. We believe a larger OSDI program is an appropriate reflection of this growth in the systems community.

We were also motivated by the challenge in making meaningful distinctions, under the pressure of program committee deadlines, between papers that are almost accepted and those almost rejected. The fragility of PC decision process has been documented and discussed elsewhere [A08]. Too often, rejections seem arbitrary in retrospect, hinging on the nuances of a PC discussion rather than clear merit. In accepting more papers we hope to incrementally improve on the fragility of these decisions, while also building a program that is more diverse and therefore of broader interest.

This goal of a larger program was a consideration throughout the review process. The PC was split into two groups: a "heavy" PC who participated in the first two rounds of reviewing, and a "heavier" PC who also reviewed papers in round three and attended a face-to-face meeting to decide final outcomes. In the first round, each paper received two reviews and approximately 35 papers were pruned. To reduce the risk of a premature pruning decision, we allowed reviewers to "rescue" a pruned paper by simply stating their support, with no discussion required. Each round-2 paper received three additional reviews. Another 80 or so papers were pruned after this round. This left us with a pool of 85 papers, each of which received two or three additional reviews in preparation for the PC meeting. After the second and third review rounds, borderline papers were discussed electronically by the reviewers and rejected by consensus of the reviewers.

In the single-day, face-to-face PC meeting each remaining paper was presented by a reviewer, generally an advocate, followed by a time-limited discussion. Based on the first discussion, we binned each paper into one of four categories: "accept," "acceptable," "questionable," and "reject." No rejects were allowed in the first part of the day, the goal of this rule being to avoid the problem of a negative start leading to rejecting good papers early. When all papers had been discussed once, we briefly considered and then accepted the "acceptable" papers as a group, then began the difficult work of reconsidering the "questionable" papers. At the end of the meeting about 30 papers had been accepted.

In the days following the PC meeting, a small set of additional papers were accepted based on an email vote by the heavier PC members. While unusual, we justified this process based on our goal to create a larger and more interesting program, and a sentiment shared by many PC members that the PC discussion had not given due consideration to several of the best liked but most controversial papers. In retrospect we believe these late accepts allowed us to create a stronger and more interesting program, and we would encourage future PC chairs to plan an appropriate process for thoughtful consideration of difficult papers after the bustle of the PC meeting has subsided. For example, even with a single-day PC meeting, it might make sense to put a small set of papers into an "overnight" category, allowing a broader collection of PC members to study them before a final decision the next day.

Apart from the review process, we took some additional measures to try and get more reviews and reviewers in a mindset to accept. We encouraged positivity, following Hill and McKinley's excellent advice [HM05]. We strictly applied conflict-of-interest rules, such that conflicted PC members were not given access to results for conflicted

papers until notifications had been sent to authors. We tried to lighten the PC load from papers that had no chance of acceptance, to leave more quality time for the remaining papers.

Before we close we'd like to briefly acknowledge a few individuals who made a difference in our bringing this program to you. The USENIX staff was fantastic throughout the entire process. We also thank Eddie Kohler for his continued support of HotCRP, a truly wonderful piece of software. We also would like to acknowledge the program committee for their tireless efforts and thoughtful reviews, and Haryadi Gunawi for his detailed note-taking during the PC meeting. Finally, we would like to thank our families and the families of PC members for supporting (and tolerating!) the long hours required to do this kind of work.

Thank you for attending OSDI '10, and have a great conference!

Remzi Arpaci-Dusseau, ***University of Wisconsin, Madison***
Brad Chen, ***Google***
OSDI '10 Program Co-Chairs

REFERENCES

[A08] "Towards a Model of Computer Systems Research," Thomas Anderson, University of Washington. WOWCS '08, April 2008.

[HM05] "Notes on Constructive and Positive Reviewing," Mark Hill, University of Wisconsin—Madison, and Kathryn S McKinley, University of Texas at Austin: http://userweb.cs.utexas.edu/users/mckinley/notes/reviewing.html, May 2005.

An Analysis of Linux Scalability to Many Cores

Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev,
M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich
MIT CSAIL

ABSTRACT

This paper analyzes the scalability of seven system applications (Exim, memcached, Apache, PostgreSQL, gmake, Psearchy, and MapReduce) running on Linux on a 48-core computer. Except for gmake, all applications trigger scalability bottlenecks inside a recent Linux kernel. Using mostly standard parallel programming techniques—this paper introduces one new technique, *sloppy counters*—these bottlenecks can be removed from the kernel or avoided by changing the applications slightly. Modifying the kernel required in total 3002 lines of code changes. A speculative conclusion from this analysis is that there is no scalability reason to give up on traditional operating system organizations just yet.

1 INTRODUCTION

There is a sense in the community that traditional kernel designs won't scale well on multicore processors: that applications will spend an increasing fraction of their time in the kernel as the number of cores increases. Prominent researchers have advocated rethinking operating systems [10, 28, 43] and new kernel designs intended to allow scalability have been proposed (e.g., Barrelfish [11], Corey [15], and fos [53]). This paper asks whether traditional kernel designs can be used and implemented in a way that allows applications to scale.

This question is difficult to answer conclusively, but we attempt to shed a small amount of light on it. We analyze scaling a number of system applications on Linux running with a 48-core machine. We examine Linux because it has a traditional kernel design, and because the Linux community has made great progress in making it scalable. The applications include the Exim mail server [2], memcached [3], Apache serving static files [1], PostgreSQL [4], gmake [23], the Psearchy file indexer [35, 48], and a multicore MapReduce library [38]. These applications, which we will refer to collectively as MOSBENCH, are designed for parallel execution and stress many major Linux kernel components.

Our method for deciding whether the Linux kernel design is compatible with application scalability is as follows. First we measure scalability of the MOSBENCH applications on a recent Linux kernel (2.6.35-rc5, released July 12, 2010) with 48 cores, using the in-memory `tmpfs` file system to avoid disk bottlenecks. gmake scales well, but the other applications scale poorly, performing much less work per core with 48 cores than with one core. We attempt to understand and fix the scalability problems, by modifying either the applications or the Linux kernel. We then iterate, since fixing one scalability problem usually exposes further ones. The end result for each application is either good scalability on 48 cores, or attribution of non-scalability to a hard-to-fix problem with the application, the Linux kernel, or the underlying hardware. The analysis of whether the kernel design is compatible with scaling rests on the extent to which our changes to the Linux kernel turn out to be modest, and the extent to which hard-to-fix problems with the Linux kernel ultimately limit application scalability.

As part of the analysis, we fixed three broad kinds of scalability problems for MOSBENCH applications: problems caused by the Linux kernel implementation, problems caused by the applications' user-level design, and problems caused by the way the applications use Linux kernel services. Once we identified a bottleneck, it typically required little work to remove or avoid it. In some cases we modified the application to be more parallel, or to use kernel services in a more scalable fashion, and in others we modified the kernel. The kernel changes are all localized, and typically involve avoiding locks and atomic instructions by organizing data structures in a distributed fashion to avoid unnecessary sharing. One reason the required changes are modest is that stock Linux already incorporates many modifications to improve scalability. More speculatively, perhaps it is the case that Linux's system-call API is well suited to an implementation that avoids unnecessary contention over kernel objects.

The main contributions of this paper are as follows. The first contribution is a set of 16 scalability improvements to the Linux 2.6.35-rc5 kernel, resulting in what we refer to as the patched kernel, PK. A few of the changes rely on a new idea, which we call *sloppy counters*, that has the nice property that it can be used to augment shared counters to make some uses more scalable without having to change all uses of the shared counter. This technique is particularly effective in Linux because typically only a few uses of a given shared counter are scalability bottlenecks; sloppy counters allow us to replace just those few uses without modifying the many other uses in the kernel. The second contribution is a set of application

benchmarks, MOSBENCH, to measure scalability of operating systems, which we make publicly available. The third is a description of the techniques required to improve the scalability of the MOSBENCH applications. Our final contribution is an analysis using MOSBENCH that suggests that there is no immediate scalability reason to give up on traditional kernel designs.

The rest of the paper is organized as follows. Section 2 relates this paper to previous work. Section 3 describes the applications in MOSBENCH and what operating system components they stress. Section 4 summarizes the differences between the stock and PK kernels. Section 5 reports on the scalability of MOSBENCH on the stock Linux 2.6.35-rc5 kernel and the PK kernel. Section 6 discusses the implications of the results. Section 7 summarizes this paper's conclusions.

2 RELATED WORK

There is a long history of work in academia and industry to scale Unix-like operating systems on shared-memory multiprocessors. Research projects such as the Stanford FLASH [33] as well as companies such as IBM, Sequent, SGI, and Sun have produced shared-memory machines with tens to hundreds processors running variants of Unix. Many techniques have been invented to scale software for these machines, including scalable locking (e.g., [41]), wait-free synchronization (e.g., [27]), multiprocessor schedulers (e.g., [8, 13, 30, 50]), memory management (e.g., [14, 19, 34, 52, 57]), and fast message passing using shared memory (e.g., [12, 47]). Textbooks have been written about adapting Unix for multiprocessors (e.g., [46]). These techniques have been incorporated in current operating systems such as Linux, Mac OS X, Solaris, and Windows. Cantrill and Bonwick summarize the historical context and real-world experience [17].

This paper extends previous scalability studies by examining a large set of systems applications, by using a 48-core PC platform, and by detailing a particular set of problems and solutions in the context of Linux. These solutions follow the standard parallel programming technique of factoring data structures so that each core can operate on separate data when sharing is not required, but such that cores can share data when necessary.

Linux scalability improvements. Early multiprocessor Linux kernels scaled poorly with kernel-intensive parallel workloads because the kernel used coarse-granularity locks for simplicity. Since then the Linux community has redesigned many kernel subsystems to improve scalability (e.g., Read-Copy-Update (RCU) [39], local run queues [6], `libnuma` [31], and improved load-balancing support [37]). The Linux symposium (`www.linuxsymposium.org`) features papers related to scalability almost every year. Some of the redesigns are based on the above-mentioned research, and some companies, such as IBM and SGI [16], have contributed code directly. Kleen provides a brief history of Linux kernel modifications for scaling and reports some areas of poor scalability in a recent Linux version (2.6.31) [32]. In this paper, we identify additional kernel scaling problems and describes how to address them.

Linux scalability studies. Gough *et al.* study the scalability of Oracle Database 10g running on Linux 2.6.18 on dual-core Intel Itanium processors [24]. The study finds problems with the Linux run queue, slab allocator, and I/O processing. Cui *et al.* uses the TPCC-UVa and Sysbench-OLTP benchmarks with PostgreSQL to study the scalability of Linux 2.6.25 on an Intel 8-core system [56], and finds application-internal bottlenecks as well as poor kernel scalability in System V IPC. We find that these problems have either been recently fixed by the Linux community or are a consequence of fixable problems in PostgreSQL.

Veal and Foong evaluate the scalability of Apache running on Linux 2.6.20.3 on an 8-core AMD Opteron computer using SPECweb2005 [51]. They identify Linux scaling problems in the kernel implementations of scheduling and directory lookup, respectively. On a 48-core computer, we also observe directory lookup as a scalability problem and PK applies a number of techniques to address this bottleneck. Pesterev *et al.* identify scalability problems in the Linux 2.6.30 network code using memcached and Apache [44]. The PK kernel addresses these problems by using a modern network card that supports a large number of virtual queues (similar to the approach taken by Route Bricks [21]).

Cui *et al.* describe microbenchmarks for measuring multicore scalability and report results from running them on Linux on a 32-core machine [55]. They find a number of scalability problems in Linux (e.g., memory-mapped file creation and deletion). Memory-mapped files show up as a scalability problem in one MOSBENCH application when multiple threads run in the same address space with memory-mapped files.

A number of new research operating systems use scalability problems in Linux as motivation. The Corey paper [15] identified bottlenecks in the Linux file descriptor and virtual memory management code caused by unnecessary sharing. Both of these bottlenecks are also triggered by MOSBENCH applications. The Barrelfish paper [11] observed that Linux TLB shootdown scales poorly. This problem is not observed in the MOSBENCH applications. Using microbenchmarks, the fos paper [53] finds that the physical page allocator in Linux 2.6.24.7 does not scale beyond 8 cores and that executing the kernel and applications on the same core results in cache interference and high miss rates. We find that the page allocator isn't a bottleneck for MOSBENCH applications on 48 cores (even though they stress memory allocation), though we have

reason to believe it would be a problem with more cores. However, the problem appears to be avoidable by, for example, using super-pages or modifying the kernel to batch page allocation.

Solaris scalability studies. Solaris provides a UNIX API and runs on SPARC-based and x86-based multicore processors. Solaris incorporates SNZIs [22], which are similar to sloppy counters (see section 4.3). Tseng *et al.* report that SAP-SD, IBM Trade and several synthetic benchmarks scale well on an 8-core SPARC system running Solaris 10 [49]. Zou *et al.* encountered coarse grained locks in the UDP networking stack of Solaris 10 that limited scalability of the OpenSER SIP proxy server on an 8-core SPARC system [29]. Using the microbenchmarks mentioned above [55], Cui *et al.* compare FreeBSD, Linux, and Solaris [54], and find that Linux scales better on some microbenchmarks and Solaris scales better on others. We ran some of the MOSBENCH applications on Solaris 10 on the 48-core machine used for this paper. While the Solaris license prohibits us from reporting quantitative results, we observed similar or worse scaling behavior compared to Linux; however, we don't know the causes or whether Solaris would perform better on SPARC hardware. We hope, however, that this paper helps others who might analyze Solaris.

3 THE MOSBENCH APPLICATIONS

To stress the kernel we chose two sets of applications: 1) applications that previous work has shown not to scale well on Linux (memcached; Apache; and Metis, a MapReduce library); and 2) applications that are designed for parallel execution and are kernel intensive (gmake, PostgreSQL, Exim, and Psearchy). Because many applications are bottlenecked by disk writes, we used an in-memory `tmpfs` file system to explore non-disk limitations. We drive some of the applications with synthetic user workloads designed to cause them to use the kernel intensively, with realism a secondary consideration. This collection of applications stresses important parts of many kernel components (e.g., the network stack, file name cache, page cache, memory manager, process manager, and scheduler). Most spend a significant fraction of their CPU time in the kernel when run on a single core. All but one encountered serious scaling problems at 48 cores caused by the stock Linux kernel. The rest of this section describes the selected applications, how they are parallelized, and what kernel services they stress.

3.1 Mail server

Exim [2] is a mail server. We operate it in a mode where a single master process listens for incoming SMTP connections via TCP and forks a new process for each connection, which in turn accepts the incoming mail, queues it in a shared set of spool directories, appends it to the per-user mail file, deletes the spooled mail, and records the delivery in a shared log file. Each per-connection process also forks twice to deliver each message. With many concurrent client connections, Exim has a good deal of parallelism. It spends 69% of its time in the kernel on a single core, stressing process creation and small file creation and deletion.

3.2 Object cache

memcached [3] is an in-memory key-value store often used to improve web application performance. A single memcached server running on multiple cores is bottlenecked by an internal lock that protects the key-value hash table. To avoid this problem, we run multiple memcached servers, each on its own port, and have clients deterministically distribute key lookups among the servers. This organization allows the servers to process requests in parallel. When request sizes are small, memcached mainly stresses the network stack, spending 80% of its time processing packets in the kernel at one core.

3.3 Web server

Apache [1] is a popular Web server, which previous work (e.g., [51]) has used to study Linux scalability. We run a single instance of Apache listening on port 80. We configure this instance to run one process per core. Each process has a thread pool to service connections; one thread is dedicated to accepting incoming connections while the other threads process the connections. In addition to the network stack, this configuration stresses the file system (in particular directory name lookup) because it stats and opens a file on every request. Running on a single core, an Apache process spends 60% of its execution time in the kernel.

3.4 Database

PostgreSQL [4] is a popular open source SQL database, which, unlike many of our other workloads, makes extensive internal use of shared data structures and synchronization. PostgreSQL also stresses many shared resources in the kernel: it stores database tables as regular files accessed concurrently by all PostgreSQL processes, it starts one process per connection, it makes use of kernel locking interfaces to synchronize and load balance these processes, and it communicates with clients over TCP sockets that share the network interface.

Ideally, PostgreSQL would scale well for read-mostly workloads, despite its inherent synchronization needs. PostgreSQL relies on snapshot isolation, a form of optimistic concurrency control that avoids most read locks. Furthermore, most write operations acquire only row-level locks exclusively and acquire all coarser-grained locks in shared modes. Thus, in principle, PostgreSQL should exhibit little contention for read-mostly workloads. In practice, PostgreSQL is limited by bottlenecks in both

its own code and in the kernel. For a read-only workload that avoids most application bottlenecks, PostgreSQL spends only 1.5% of its time in the kernel with one core, but this grows to 82% with 48 cores.

3.5 Parallel build

gmake [23] is an implementation of the standard make utility that supports executing independent build rules concurrently. gmake is the unofficial default benchmark in the Linux community since all developers use it to build the Linux kernel. Indeed, many Linux patches include comments like "This speeds up compiling the kernel." We benchmarked gmake by building the stock Linux 2.6.35-rc5 kernel with the default configuration for x86_64. gmake creates more processes than there are cores, and reads and writes many files. The execution time of gmake is dominated by the compiler it runs, but system time is not negligible: with one core, 7.6% of the execution time is system time.

3.6 File indexer

Psearchy is a parallel version of searchy [35, 48], a program to index and query Web pages. We focus on the indexing component of searchy because it is more system intensive. Our parallel version, pedsort, runs the searchy indexer on each core, sharing a work queue of input files. Each core operates in two phases. In phase 1, it pulls input files off the work queue, reading each file and recording the positions of each word in a per-core hash table. When the hash table reaches a fixed size limit, it sorts it alphabetically, flushes it to an intermediate index on disk, and continues processing input files. Phase 1 is both compute intensive (looking up words in the hash table and sorting it) and file-system intensive (reading input files and flushing the hash table). To avoid stragglers in phase 1, the initial work queue is sorted so large files are processed first. Once the work queue is empty, each core merges the intermediate index files it produced, concatenating the position lists of words that appear in multiple intermediate indexes, and generates a binary file that records the positions of each word and a sequence of Berkeley DB files that map each word to its byte offset in the binary file. To simplify the scalability analysis, each core starts a new Berkeley DB every 200,000 entries, eliminating a logarithmic factor and making the aggregate work performed by the indexer constant regardless of the number of cores. Unlike phase 1, phase 2 is mostly file-system intensive. While pedsort spends only 1.9% of its time in the kernel at one core, this grows to 23% at 48 cores, indicating scalability limitations.

3.7 MapReduce

Metis is a MapReduce [20] library for single multicore servers inspired by Phoenix [45]. We use Metis with an application that generates inverted indices. This workload allocates large amounts of memory to hold temporary tables, stressing the kernel memory allocator and soft page fault code. This workload spends 3% of its runtime in the kernel with one core, but this rises to 16% at 48 cores.

4 KERNEL OPTIMIZATIONS

The MOSBENCH applications trigger a few scalability bottlenecks in the kernel. We describe the bottlenecks and our solutions here, before presenting detailed per-application scaling results in Section 5, because many of the bottlenecks are common to multiple applications. Figure 1 summarizes the bottlenecks. Some of these problems have been discussed on the Linux kernel mailing list and solutions proposed; perhaps the reason these solutions have not been implemented in the standard kernel is that the problems are not acute on small-scale SMPs or are masked by I/O delays in many applications. Figure 1 also summarizes our solution for each bottleneck.

4.1 Scalability tutorial

Why might one expect performance to scale well with the number of cores? If a workload consists of an unlimited supply of tasks that do not interact, then you'd expect to get linear increases in total throughput by adding cores and running tasks in parallel. In real life parallel tasks usually interact, and interaction usually forces serial execution. Amdahl's Law summarizes the result: however small the serial portion, it will eventually prevent added cores from increasing performance. For example, if 25% of a program is serial (perhaps inside some global locks), then any number of cores can provide no more than 4-times speedup.

Here are a few types of serializing interactions that the MOSBENCH applications encountered. These are all classic considerations in parallel programming, and are discussed in previous work such as [17].

- The tasks may lock a shared data structure, so that increasing the number of cores increases the lock wait time.
- The tasks may write a shared memory location, so that increasing the number of cores increases the time spent waiting for the cache coherence protocol to fetch the cache line in exclusive mode. This problem can occur even in lock-free shared data structures.
- The tasks may compete for space in a limited-size shared hardware cache, so that increasing the number of cores increases the cache miss rate. This problem can occur even if tasks never share memory.
- The tasks may compete for other shared hardware resources such as inter-core interconnect or DRAM

Parallel `accept`			Apache
Concurrent `accept` system calls contend on shared `socket` fields.	⇒	User per-core backlog queues for listening sockets.	
`dentry` reference counting			Apache, Exim
File name resolution contends on directory entry reference counts.	⇒	Use sloppy counters to reference count directory entry objects.	
Mount point (`vfsmount`) reference counting			Apache, Exim
Walking file name paths contends on mount point reference counts.	⇒	Use sloppy counters for mount point objects.	
IP packet destination (`dst_entry`) reference counting			memcached, Apache
IP packet transmission contends on routing table entries.	⇒	Use sloppy counters for IP routing table entries.	
Protocol memory usage tracking			memcached, Apache
Cores contend on counters for tracking protocol memory consumption.	⇒	Use sloppy counters for protocol usage counting.	
Acquiring directory entry (`dentry`) spin locks			Apache, Exim
Walking file name paths contends on per-directory entry spin locks.	⇒	Use a lock-free protocol in `dlookup` for checking filename matches.	
Mount point table spin lock			Apache, Exim
Resolving path names to mount points contends on a global spin lock.	⇒	Use per-core mount table caches.	
Adding files to the open list			Apache, Exim
Cores contend on a per-super block list that tracks open files.	⇒	Use per-core open file lists for each super block that has open files.	
Allocating DMA buffers			memcached, Apache
DMA memory allocations contend on the memory node 0 spin lock.	⇒	Allocate Ethernet device DMA buffers from the local memory node.	
False sharing in `net_device` and `device`			memcached, Apache, PostgreSQL
False sharing causes contention for read-only structure fields.	⇒	Place read-only fields on their own cache lines.	
False sharing in `page`			Exim
False sharing causes contention for read-mostly structure fields.	⇒	Place read-only fields on their own cache lines.	
`inode` lists			memcached, Apache
Cores contend on global locks protecting lists used to track `inode`s.	⇒	Avoid acquiring the locks when not necessary.	
Dcache lists			memcached, Apache
Cores contend on global locks protecting lists used to track `dentry`s.	⇒	Avoid acquiring the locks when not necessary.	
Per-`inode` mutex			PostgreSQL
Cores contend on a per-`inode` mutex in `lseek`.	⇒	Use atomic reads to eliminate the need to acquire the mutex.	
Super-page fine grained locking			Metis
Super-page soft page faults contend on a per-process mutex.	⇒	Protect each super-page memory mapping with its own mutex.	
Zeroing super-pages			Metis
Zeroing super-pages flushes the contents of on-chip caches.	⇒	Use non-caching instructions to zero the contents of super-pages.	

Figure 1: A summary of Linux scalability problems encountered by MOSBENCH applications and their corresponding fixes. The fixes add 2617 lines of code to Linux and remove 385 lines of code from Linux.

interfaces, so that additional cores spend their time waiting for those resources rather than computing.

- There may be too few tasks to keep all cores busy, so that increasing the number of cores leads to more idle cores.

Many scaling problems manifest themselves as delays caused by cache misses when a core uses data that other cores have written. This is the usual symptom both for lock contention and for contention on lock-free mutable data. The details depend on the hardware cache coherence protocol, but the following is typical. Each core has a data cache for its own use. When a core writes data that other cores have cached, the cache coherence protocol forces the write to wait while the protocol finds the cached copies and invalidates them. When a core reads data that another core has just written, the cache coherence protocol doesn't return the data until it finds the cache that holds the modified data, annotates that cache to indicate there is a copy of the data, and fetches the data to the reading core. These operations take about the same time as loading data from off-chip RAM (hundreds of cycles), so sharing mutable data can have a disproportionate effect on performance.

Exercising the cache coherence machinery by modifying shared data can produce two kinds of scaling problems. First, the cache coherence protocol serializes modifications to the same cache line, which can prevent parallel speedup. Second, in extreme cases the protocol may saturate the inter-core interconnect, again preventing additional cores from providing additional performance. Thus good performance and scalability often demand that data be structured so that each item of mutable data is used by only one core.

In many cases scaling bottlenecks limit performance to some maximum, regardless of the number of cores. In other cases total throughput decreases as the number of cores grows, because each waiting core slows down the cores that are making progress. For example, non-scalable spin locks produce per-acquire interconnect traffic that is proportional to the number of waiting cores; this traffic may slow down the core that holds the lock by an amount proportional to the number of waiting cores [41]. Acquir-

ing a Linux spin lock takes a few cycles if the acquiring core was the previous lock holder, takes a few hundred cycles if another core last held the lock and there is no contention, and are not scalable under contention.

Performance is often the enemy of scaling. One way to achieve scalability is to use inefficient algorithms, so that each core busily computes and makes little use of shared resources such as locks. Conversely, increasing the efficiency of software often makes it less scalable, by increasing the fraction of time it uses shared resources. This effect occurred many times in our investigations of MOSBENCH application scalability.

Some scaling bottlenecks cannot easily be fixed, because the semantics of the shared resource require serial access. However, it is often the case that the implementation can be changed so that cores do not have to wait for each other. For example, in the stock Linux kernel the set of runnable threads is partitioned into mostly-private per-core scheduling queues; in the common case, each core only reads, writes, and locks its own queue [36]. Many scaling modifications to Linux follow this general pattern.

Many of our scaling modifications follow this same pattern, avoiding both contention for locks and contention for the underlying data. We solved other problems using well-known techniques such as lock-free protocols or fine-grained locking. In all cases we were able to eliminate scaling bottlenecks with only local changes to the kernel code. The following subsections explain our techniques.

4.2 Multicore packet processing

The Linux network stack connects different stages of packet processing with queues. A received packet typically passes through multiple queues before finally arriving at a per-socket queue, from which the application reads it with a system call like `read` or `accept`. Good performance with many cores and many independent network connections demands that each packet, queue, and connection be handled by just one core [21, 42]. This avoids inter-core cache misses and queue locking costs.

Recent Linux kernels take advantage of network cards with multiple hardware queues, such as Intel's 82599 10Gbit Ethernet (IXGBE) card, or use software techniques, such as Receive Packet Steering [26] and Receive Flow Steering [25], to attempt to achieve this property. With a multi-queue card, Linux can be configured to assign each hardware queue to a different core. Transmit scaling is then easy: Linux simply places outgoing packets on the hardware queue associated with the current core. For incoming packets, such network cards provide an interface to configure the hardware to enqueue incoming packets matching a particular criteria (e.g., source IP address and port number) on a specific queue and thus to a particular core. This spreads packet processing load across cores. However, the IXGBE driver goes further: for each core, it samples every 20^{th} outgoing TCP packet and updates the hardware's flow directing tables to deliver further incoming packets from that TCP connection directly to the core.

This design typically performs well for long-lived connections, but poorly for short ones. Because the technique is based on sampling, it is likely that the majority of packets on a given short connection will be misdirected, causing cache misses as Linux delivers to the socket on one core while the socket is used on another. Furthermore, because few packets are received per short-lived connection, misdirecting even the initial handshake packet of a connection imposes a significant cost.

For applications like Apache that simultaneously accept connections on all cores from the same listening socket, we address this problem by allowing the hardware to determine which core and thus which application thread will handle an incoming connection. We modify `accept` to prefer connections delivered to the local core's queue. Then, if the application processes the connection on the same core that accepted it (as in Apache), all processing for that connection will remain entirely on one core. Our solution has the added benefit of addressing contention on the lock that protects the single listening socket's connection backlog queue.

To implement this, we configured the IXGBE to direct each packet to a queue (and thus core) using a hash of the packet headers designed to deliver all of a connection's packets (including the TCP handshake packets) to the same core. We then modified the code that handles TCP connection setup requests to queue requests on a per-core backlog queue for the listening socket, so that a thread will accept and process connections that the IXGBE directs to the core running that thread. If `accept` finds the current core's backlog queue empty, it attempts to steal a connection request from a different core's queue. This arrangement provides high performance for short connections by processing each connection entirely on one core. If threads were to move from core to core while handling a single connection, a combination of this technique and the current sampling approach might be best.

4.3 Sloppy counters

Linux uses shared counters for reference-counted garbage collection and to manage various resources. These counters can become bottlenecks if many cores update them. In these cases lock-free atomic increment and decrement instructions do not help, because the coherence hardware serializes the operations on a given counter.

The MOSBENCH applications encountered bottlenecks from reference counts on directory entry objects (`dentry`s), mounted file system objects (`vfsmount`s), network routing table entries (`dst_entry`s), and counters

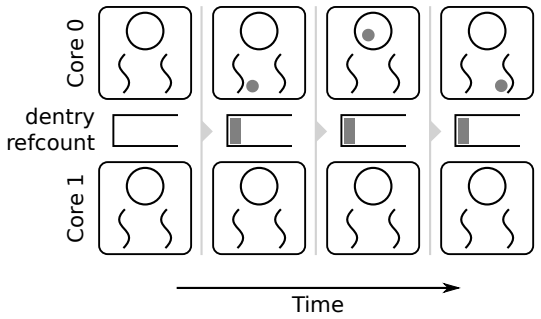


Figure 2: An example of the kernel using a sloppy counter for `dentry` reference counting. A large circle represents a local counter, and a gray dot represents a held reference. In this figure, a thread on core 0 first acquires a reference from the central counter. When the thread releases this reference, it adds the reference to the local counter. Finally, another thread on core 0 is able to acquire the spare reference without touching the central counter.

tracking the amount of memory allocated by each network protocol (such as TCP or UDP).

Our solution, which we call *sloppy counters*, builds on the intuition that each core can hold a few spare references to an object, in hopes that it can give ownership of these references to threads running on that core, without having to modify the global reference count. More concretely, a sloppy counter represents one logical counter as a single shared central counter and a set of per-core counts of spare references. When a core increments a sloppy counter by V, it first tries to acquire a spare reference by *decrementing* its per-core counter by V. If the per-core counter is greater than or equal to V, meaning there are sufficient local references, the decrement succeeds. Otherwise the core must acquire the references from the central counter, so it increments the shared counter by V. When a core decrements a sloppy counter by V, it releases these references as local spare references, *incrementing* its per-core counter by V. Figure 2 illustrates incrementing and decrementing a sloppy counter. If the local count grows above some threshold, spare references are released by decrementing both the per-core count and the central count.

Sloppy counters maintain the invariant that the sum of per-core counters and the number of resources in use equals the value in the shared counter. For example, a shared `dentry` reference counter equals the sum of the per-core counters and the number of references to the `dentry` currently in use.

A core usually updates a sloppy counter by modifying its per-core counter, an operation which typically only needs to touch data in the core's local cache (no waiting for locks or cache-coherence serialization).

We added sloppy counters to count references to `dentry`s, `vfsmount`s, and `dst_entry`s, and used sloppy counters to track the amount of memory allocated by each network protocol (such as TCP and UDP). Only uses of a counter that cause contention need to be modified, since sloppy counters are backwards-compatible with existing shared-counter code. The kernel code that creates a sloppy counter allocates the per-core counters. It is occasionally necessary to reconcile the central and per-core counters, for example when deciding whether an object can be de-allocated. This operation is expensive, so sloppy counters should only be used for objects that are relatively infrequently de-allocated.

Sloppy counters are similar to Scalable NonZero Indicators (SNZI) [22], distributed counters [9], and approximate counters [5]. All of these techniques speed up increment/decrement by use of per-core counters, and require significantly more work to find the true total value. Sloppy counters are attractive when one wishes to improve the performance of some uses of an existing counter without having to modify all points in the code where the counter is used. A limitation of sloppy counters is that they use space proportional to the number of cores.

4.4 Lock-free comparison

We found situations in which MOSBENCH applications were bottlenecked by low scalability for name lookups in the directory entry cache. The directory entry cache speeds up lookups by mapping a directory and a file name to a `dentry` identifying the target file's `inode`. When a potential `dentry` is located, the lookup code acquires a per-`dentry` spin lock to atomically compare several fields of the `dentry` with the arguments of the lookup function. Even though the directory cache has been optimized using RCU for scalability [40], the `dentry` spin lock for common parent directories, such as `/usr`, was sometimes a bottleneck even if the path names ultimately referred to different files.

We optimized `dentry` comparisons using a lock-free protocol similar to Linux' lock-free page cache lookup protocol [18]. The lock-free protocol uses a generation counter, which the PK kernel increments after every modification to a directory entry (e.g., `mv foo bar`). During a modification (when the `dentry` spin lock is held), PK temporarily sets the generation counter to 0. The PK kernel compares `dentry` fields to the arguments using the following procedure for atomicity:

- If the generation counter is 0, fall back to the locking protocol. Otherwise remember the value of the generation counter.
- Copy the fields of the `dentry` to local variables. If the generation afterwards differs from the remembered value, fall back to the locking protocol.
- Compare the copied fields to the arguments. If there is a match, increment the reference count unless it is 0, and return the `dentry`. If the reference count is 0, fall back to the locking protocol.

The lock-free protocol improves scalability because it allows cores to perform lookups for the same directory entries without serializing.

4.5 Per-core data structures

We encountered three kernel data structures that caused scaling bottlenecks due to lock contention: a per-super-block list of open files that determines whether a read-write file system can be remounted read-only, a table of mount points used during path lookup, and the pool of free packet buffers. Though each of these bottlenecks is caused by lock contention, bottlenecks would remain if we replaced the locks with finer grained locks or a lock free protocol, because multiple cores update the data structures. Therefore our solutions refactor the data structures so that in the common case each core uses different data.

We split the per-super-block list of open files into per-core lists. When a process opens a file the kernel locks the current core's list and adds the file. In most cases a process closes the file on the same core it opened it on. However, the process might have migrated to another core, in which case the file must be expensively removed from the list of the original core. When the kernel checks if a file system can be remounted read-only it must lock and scan all cores' lists.

We also added per-core `vfsmount` tables, each acting as a cache for a central `vfsmount` table. When the kernel needs to look up the `vfsmount` for a path, it first looks in the current core's table, then the central table. If the latter succeeds, the result is added to the per-core table.

Finally, the default Linux policy for machines with NUMA memory is to allocate packet buffers (`skbuff`s) from a single free list in the memory system closest to the I/O bus. This caused contention for the lock protecting the free list. We solved this using per-core free lists.

4.6 Eliminating false sharing

We found some MOSBENCH applications caused false sharing in the kernel. In the cases we identified, the kernel located a variable it updated often on the same cache line as a variable it read often. The result was that cores contended for the falsely shared line, limiting scalability. Exim per-core performance degraded because of false sharing of physical page reference counts and flags, which the kernel located on the same cache line of a `page` variable. memcached, Apache, and PostgreSQL faced similar false sharing problems with `net_device` and `device` variables. In all cases, placing the heavily modified data on a separate cache line improved scalability.

4.7 Avoiding unnecessary locking

For small numbers of cores, lock contention in Linux does not limit scalability for MOSBENCH applications. With more than 16 cores, the scalability of memcached, Apache, PostgreSQL, and Metis are limited by waiting for and acquiring spin locks and mutexes[1] in the file system and virtual memory management code. In many cases we were able to eliminate acquisitions of the locks altogether by modifying the code to detect special cases when acquiring the locks was unnecessary. In one case, we split a mutex protecting all the super page mappings into one mutex per mapping.

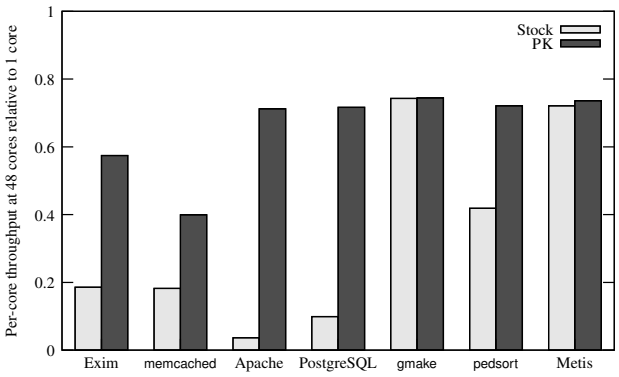


Figure 3: MOSBENCH results summary. Each bar shows the ratio of per-core throughput with 48 cores to throughput on one core, with 1.0 indicating perfect scalability. Each pair of bars corresponds to one application before and after our kernel and application modifications.

5 EVALUATION

This section evaluates the MOSBENCH applications on the most recent Linux kernel at the time of writing (Linux 2.6.35-rc5, released on July 12, 2010) and our modified version of this kernel, PK. For each application, we describe how the stock kernel limits scalability, and how we addressed the bottlenecks by modifying the application and taking advantage of the PK changes.

Figure 3 summarizes the results of the MOSBENCH benchmark, comparing application scalability before and after our modifications. A bar with height 1.0 indicates perfect scalability (48 cores yielding a speedup of 48). Most of the applications scale significantly better with our modifications. All of them fall short of perfect scalability even with those modifications. As the rest of this section explains, the remaining scalability bottlenecks are not the fault of the kernel. Instead, they are caused by non-parallelizable components in the application or underlying hardware: resources that the application's design requires it to share, imperfect load balance, or hardware bottlenecks such as the memory system or the network card. For this reason, we conclude that the Linux kernel with our modifications is consistent with MOSBENCH scalability up to 48 cores.

For each application we show scalability plots in the same format, which shows throughput per core (see, for example, Figure 4). A horizontal line indicates perfect

[1] A thread initially busy waits to acquire a mutex, but if the wait time is long the thread yields the CPU.

scalability: each core contributes the same amount of work regardless of the total number of cores. In practice one cannot expect a truly horizontal line: a single core usually performs disproportionately well because there is no inter-core sharing and because Linux uses a streamlined lock scheme with just one core, and the per-chip caches become less effective as more active cores share them. For most applications we see the stock kernel's line drop sharply because of kernel bottlenecks, and the PK line drop more modestly.

5.1 Method

We run the applications that modify files on a `tmpfs` in-memory file system to avoid waiting for disk I/O. The result is that MOSBENCH stresses the kernel more it would if it had to wait for the disk, but that the results are not representative of how the applications would perform in a real deployment. For example, a real mail server would probably be bottlenecked by the need to write each message durably to a hard disk. The purpose of these experiments is to evaluate the Linux kernel's multicore performance, using the applications to generate a reasonably realistic mix of system calls.

We run experiments on a 48-core machine, with a Tyan Thunder S4985 board and an M4985 quad CPU daughterboard. The machine has a total of eight 2.4 GHz 6-core AMD Opteron 8431 chips. Each core has private 64 Kbyte instruction and data caches, and a 512 Kbyte private L2 cache. The cores on each chip share a 6 Mbyte L3 cache, 1 Mbyte of which is used for the HT Assist probe filter [7]. Each chip has 8 Gbyte of local off-chip DRAM. A core can access its L1 cache in 3 cycles, its L2 cache in 14 cycles, and the shared on-chip L3 cache in 28 cycles. DRAM access latencies vary, from 122 cycles for a core to read from its local DRAM to 503 cycles for a core to read from the DRAM of the chip farthest from it on the interconnect. The machine has a dual-port Intel 82599 10Gbit Ethernet (IXGBE) card, though we use only one port for all experiments. That port connects to an Ethernet switch with a set of load-generating client machines.

Experiments that use fewer than 48 cores run with the other cores entirely disabled. memcached, Apache, Psearchy, and Metis pin threads to cores; the other applications do not. We run each experiment 3 times and show the best throughput, in order to filter out unrelated activity; we found the variation to be small.

5.2 Exim

To measure the performance of Exim 4.71, we configure Exim to use `tmpfs` for all mutable files—spool files, log files, and user mail files—and disable DNS and RFC1413 lookups. Clients run on the same machine as Exim. Each repeatedly opens an SMTP connection to Exim, sends 10 separate 20-byte messages to a local user, and closes the SMTP connection. Sending 10 messages per connection

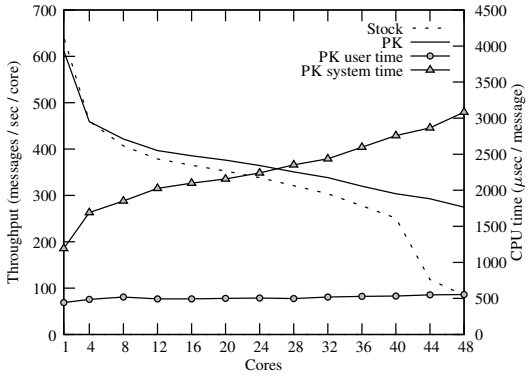


Figure 4: Exim throughput and runtime breakdown.

prevents exhaustion of TCP client port numbers. Each client sends to a different user to prevent contention on user mail files. We use 96 client processes regardless of the number of active cores; as long as there are enough clients to keep Exim busy, the number of clients has little effect on performance.

We modified and configured Exim to increase performance on both the stock and PK kernels:

- Berkeley DB v4.6 reads /proc/stat to find the number of cores. This consumed about 20% of the total runtime, so we modified Berkeley DB to aggressively cache this information.
- We configured Exim to split incoming queued messages across 62 spool directories, hashing by the per-connection process ID. This improves scalability because delivery processes are less likely to create files in the same directory, which decreases contention on the directory metadata in the kernel.
- We configured Exim to avoid an `exec()` per mail message, using `deliver_drop_privilege`.

Figure 4 shows the number of messages Exim can process per second on each core, as the number of cores varies. The stock and PK kernels perform nearly the same on one core. As the number of cores increases, the per-core throughput of the stock kernel eventually drops toward zero. The primary cause of the throughput drop is contention on a non-scalable kernel spin lock that serializes access to the `vfsmount` table. Exim causes the kernel to access the `vfsmount` table dozens of times for each message. Exim on PK scales significantly better, owing primarily to improvements to the `vfsmount` table (Section 4.5) and the changes to the `dentry` cache (Section 4.4).

Throughput on the PK kernel degrades from one to two cores, while the system time increases, because of the many kernel data structures that are not shared with one core but must be shared (with cache misses) with

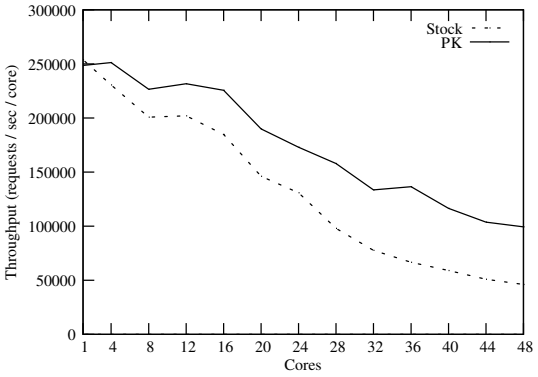


Figure 5: memcached throughput.

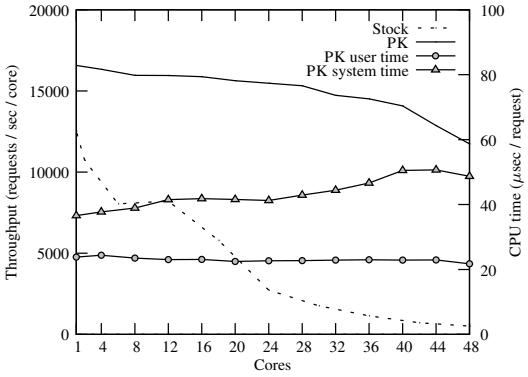


Figure 6: Apache throughput and runtime breakdown.

two cores. The throughput on the PK kernel continues to degrade; however, this is mainly due to application-induced contention on the per-directory locks protecting file creation in the spool directories. As the number of cores increases, there is an increasing probability that Exim processes running on different cores will choose the same spool directory, resulting in the observed contention.

We foresee a potential bottleneck on more cores due to cache misses when a per-connection process and the delivery process it forks run on different cores. When this happens the delivery process suffers caches misses when it first accesses kernel data—especially data related to virtual address mappings—that its parent initialized. The result is that process destruction, which frees virtual address mappings, and soft page fault handling, which reads virtual address mappings, execute more slowly with more cores. For the Exim configuration we use, however, this slow down is negligible compared to slow down that results from contention on spool directories.

5.3 memcached

We run a separate memcached 1.4.4 process on each core to avoid application lock contention. Each server is pinned to a separate core and has its own UDP port. Each client thread repeatedly queries a particular memcached instance for a non-existent key because this places higher load on the kernel than querying for existing keys. There are a total of 792 client threads running on 22 client machines. Requests are 68 bytes, and responses are 64. Each client thread sends a batch of 20 requests and waits for the responses, timing out after 100 ms in case packets are lost.

For both kernels, we use a separate hardware receive and transmit queue for each core and configure the IXGBE to inspect the port number in each incoming packet header, place the packet on the queue dedicated to the associated memcached's core, and deliver the receive interrupt to that core.

Figure 5 shows that memcached does not scale well on the stock Linux kernel.

One scaling problem occurs in the memory allocator. Linux associates a separate allocator with each socket to allocate memory from that chip's attached DRAM. The stock kernel allocates each packet from the socket nearest the PCI bus, resulting in contention on that socket's allocator. We modified the allocation policy to allocate from the local socket, which improved throughput by ∼30%.

Another bottleneck was false read/write sharing of IXGBE device driver data in the `net_device` and `device` structures, resulting in cache misses for all cores even on read-only fields. We rearranged both structures to isolate critical read-only members to their own cache lines. Removing a single falsely shared cache line in `net_device` increased throughput by 30% at 48 cores.

The final bottleneck was contention on the `dst_entry` structure's reference count in the network stack's destination cache, which we replaced with a sloppy counter (see Section 4.3).

The "PK" line in Figure 5 shows the scalability of memcached with these changes. The per core throughput drops off after 16 cores. We have isolated this bottleneck to the IXGBE card itself, which appears to handle fewer packets as the number of virtual queues increases. As a result, it fails to transmit packets at line rate even though there are always packets queued in the DMA rings.

To summarize, while memcached scales poorly, the bottlenecks caused by the Linux kernel were fixable and the remaining bottleneck lies in the hardware rather than in the Linux kernel.

5.4 Apache

A single instance of Apache running on stock Linux scales very poorly because of contention on a mutex protecting the single accept socket. Thus, for stock Linux, we run a separate instance of Apache per core with each server running on a distinct port. Figure 6 shows that Apache still scales poorly on the stock kernel, even with separate Apache instances.

For PK, we run a single instance of Apache 2.2.14 on one TCP port. Apache serves a single static file from an

ext3 file system; the file resides in the kernel buffer cache. We serve a file that is 300 bytes because transmitting a larger file exhausts the available 10 Gbit bandwidth at a low server core count. Each request involves accepting a TCP connection, opening the file, copying its content to a socket, and closing the file and socket; logging is disabled. We use 58 client processes running on 25 physical client machines (many clients are themselves multi-core). For each active server core, each client opens 2 TCP connections to the server at a time (so, for a 48-core server, each client opens 96 TCP connections).

All the problems and solutions described in Section 5.3 apply to Apache, as do the modifications to the `dentry` cache for both files and sockets described in Section 4. Apache forks off a process per core, pinning each new process to a different core. Each process dedicates a thread to accepting connections from the shared listening socket and thus, with the accept queue changes described in Section 4.2, each connection is accepted on the core it initially arrives on and all packet processing is performed local to that core. The PK numbers in Figure 6 are significantly better than Apache running on the stock kernel; however, Apache's throughput on PK does not scale linearly.

Past 36 cores, performance degrades because the network card cannot keep up with the increasing workload. Lack of work causes the server idle time to reach 18% at 48 cores. At 48 cores, the network card's internal diagnostic counters show that the card's internal receive packet FIFO overflows. These overflows occur even though the clients are sending a total of only 2 Gbits and 2.8 million packets per second when other independent tests have shown that the card can either receive upwards of 4 Gbits per second or process 5 million packets per second.

We created a microbenchmark that replicates the Apache network workload, but uses substantially less CPU time on the server. In the benchmark, the client machines send UDP packets as fast as possible to the server, which also responds with UDP packets. The packet mix is similar to that of the Apache benchmark. While the microbenchmark generates far more packets than the Apache clients, the network card ultimately delivers a similar number of packets per second as in the Apache benchmark and drops the rest. Thus, at high core counts, the network card is unable to deliver additional load to Apache, which limits its scalability.

5.5 PostgreSQL

We evaluate Linux's scalability running PostgreSQL 8.3.9 using both a 100% read workload and a 95%/5% read/write workload. The database consists of a single indexed 600 Mbyte table of 10,000,000 key-value pairs stored in `tmpfs`. We configure PostgreSQL to use a 2 Gbyte application-level cache because PostgreSQL protects its cache free-list with a single lock and thus

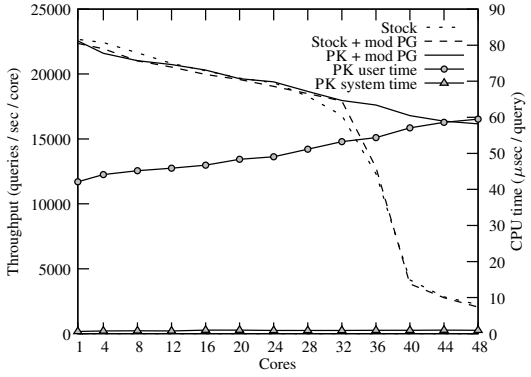


Figure 7: PostgreSQL read-only workload throughput and runtime breakdown.

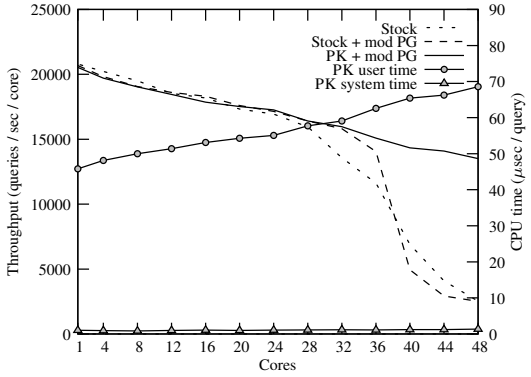


Figure 8: PostgreSQL read/write workload throughput and runtime breakdown.

scales poorly with smaller caches. While we do not pin the PostgreSQL processes to cores, we do rely on the IXGBE driver to route packets from long-lived connections directly to the cores processing those connections.

Our workload generator simulates typical high-performance PostgreSQL configurations, where middleware on the client machines aggregates multiple client connections into a small number of connections to the server. Our workload creates one PostgreSQL connection per server core and sends queries (selects or updates) in batches of 256, aggregating successive read-only transactions into single transactions. This workload is intended to minimize application-level contention within PostgreSQL in order to maximize the stress PostgreSQL places on the kernel.

The "Stock" line in Figures 7 and 8 shows that PostgreSQL has poor scalability on the stock kernel. The first bottleneck we encountered, which caused the read/write workload's total throughput to peak at only 28 cores, was due to PostgreSQL's design. PostgreSQL implements row- and table-level locks atop user-level mutexes; as a result, even a non-conflicting row- or table-level lock acquisition requires exclusively locking one of only 16 global mutexes. This leads to unnecessary contention for non-conflicting acquisitions of the same lock—as seen in

the read/write workload—and to false contention between unrelated locks that hash to the same exclusive mutex. We address this problem by rewriting PostgreSQL's row- and table-level lock manager and its mutexes to be lock-free in the uncontended case, and by increasing the number of mutexes from 16 to 1024.

The "Stock + mod PG" line in Figures 7 and 8 shows the results of this modification, demonstrating improved performance out to 36 cores for the read/write workload. While performance still collapses at high core counts, the cause of this has shifted from excessive user time to excessive system time. The read-only workload is largely unaffected by the modification as it makes little use of row- and table-level locks.

With modified PostgreSQL on stock Linux, throughput for both workloads collapses at 36 cores, with system time rising from 1.7 μseconds/query at 32 cores to 322 μseconds/query at 48 cores. The main reason is the kernel's `lseek` implementation. PostgreSQL calls `lseek` many times per query on the same two files, which in turn acquires a mutex on the corresponding `inode`. Linux's adaptive mutex implementation suffers from starvation under intense contention, resulting in poor performance. However, the mutex acquisition turns out not to be necessary, and PK eliminates it.

Figures 7 and 8 show that, with PK's modified `lseek` and smaller contributions from other PK changes, PostgreSQL performance no longer collapses. On PK, PostgreSQL's overall scalability is primarily limited by contention for the spin lock protecting the buffer cache page for the root of the table index. It spends little time in the kernel, and is not limited by Linux's performance.

5.6 gmake

We measure the performance of parallel gmake by building the object files of Linux 2.6.35-rc5 for x86_64. All input source files reside in the buffer cache, and the output files are written to `tmpfs`. We set the maximum number of concurrent jobs of gmake to twice the number of cores.

Figure 9 shows that gmake on 48 cores achieves excellent scalability, running 35 times faster on 48 cores than on one core for both the stock and PK kernels. The PK kernel shows slightly lower system time owing to the changes to the `dentry` cache. gmake scales imperfectly because of serial stages at the beginning of the build and straggling processes at the end.

gmake scales so well in part because much of the CPU time is in the compiler, which runs independently on each core. In addition, Linux kernel developers have thoroughly optimized kernel compilation, since it is of particular importance to them.

5.7 Psearchy/pedsort

Figure 10 shows the runtime for different versions of pedsort indexing the Linux 2.6.35-rc5 source tree, which

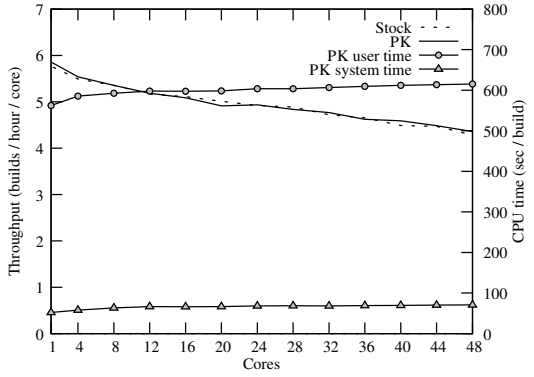


Figure 9: gmake throughput and runtime breakdown.

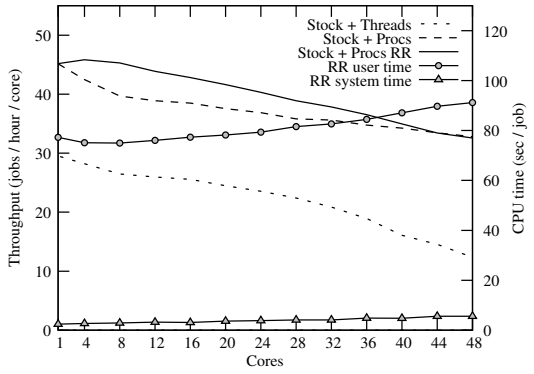


Figure 10: pedsort throughput and runtime breakdown.

consists of 368 Mbyte of text across 33,312 source files. The input files are in the buffer cache and the output files are written to `tmpfs`. Each core uses a 48 Mbyte word hash table and limits the size of each output index to 200,000 entries (see Section 3.6). As a result, the total work performed by pedsort and its final output are independent of the number of cores involved.

The initial version of pedsort used a single process with one thread per core. The line marked "Stock + Threads" in Figure 10 shows that it scales badly. Most of the increase in runtime is in system time: for 1 core the system time is 2.3 seconds, while at 48 cores the total system time is 41 seconds.

Threaded pedsort scales poorly because a per-process kernel mutex serializes calls to `mmap` and `munmap` for a process' virtual address space. pedsort reads input files using `libc` file streams, which access file contents via `mmap`, resulting in contention over the shared address space, even though these memory-mapped files are logically private to each thread in pedsort. We avoided this problem by modifying pedsort to use one process per core for concurrency, eliminating the `mmap` contention by eliminating the shared address space. This modification involved changing about 10 lines of code in pedsort. The performance of this version on the stock kernel is shown as "Stock + Procs" in Figure 10. Even on a single core,

the multi-process version outperforms the threaded version because any use of threads forces glibc to use slower, thread-safe variants of various library functions.

With a small number of cores, the performance of the process version depends on how many cores share the per-socket L3 caches. Figure 10's "Stock + Procs" line shows performance when the active cores are spread over few sockets, while the "Stock + Procs RR" shows performance when the active cores are spread evenly over sockets. As corroborated by hardware performance counters, the latter scheme provides higher performance because each new socket provides access to more total L3 cache space.

Using processes, system time remains small, so the kernel is not a limiting factor. Rather, as the number of cores increases, pedsort spends more time in the glibc sorting function `msort_with_tmp`, which causes the decreasing throughput and rising user time in Figure 10. As the number of cores increases and the total working set size per socket grows, `msort_with_tmp` experiences higher L3 cache miss rates. However, despite its memory demands, `msort_with_tmp` never reaches the DRAM bandwidth limit. Thus, pedsort is bottlenecked by cache capacity.

5.8 Metis

We measured Metis performance by building an inverted index from a 2 Gbyte in-memory file. As for Psearchy, we spread the active cores across sockets and thus have access to the machine's full L3 cache space at 8 cores.

The "Stock + 4 KB pages" line in Figure 11 shows Metis' original performance. As the number of cores increases, the per-core performance of Metis decreases. Metis allocates memory with `mmap`, which adds the new memory to a region list but defers modifying page tables. When a fault occurs on a new mapping, the kernel locks the entire region list with a read lock. When many concurrent faults occur on different cores, the lock itself becomes a bottleneck, because acquiring it even in read mode involves modifying shared lock state.

We avoided this problem by mapping memory with 2 Mbyte *super-pages*, rather than 4 Kbyte pages, using Linux's `hugetlbfs`. This results in many fewer page faults and less contention on the region list lock. We also used finer-grained locking in place of a global mutex that serialized super-page faults. The "PK + 2MB pages" line in Figure 11 shows that use of super-pages increases performance and significantly reduces system time.

With super-pages, the time spent in the kernel becomes negligible and Metis' scalability is limited primarily by the DRAM bandwidth required by the reduce phase. This phase is particularly memory-intensive and, at 48 cores, accesses DRAM at 50.0 Gbyte/second, just shy of the maximum achievable throughput of 51.5 Gbyte/second measured by our microbenchmarks.

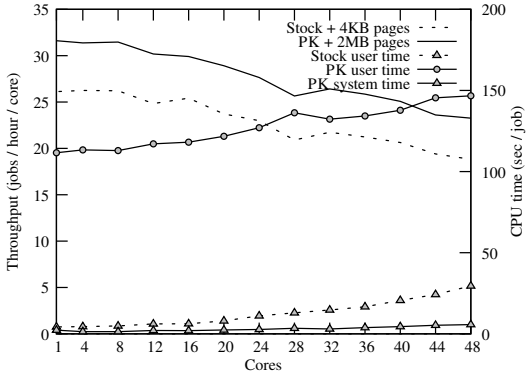


Figure 11: Metis throughput and runtime breakdown.

Application	Bottleneck
Exim	App: Contention on spool directories
memcached	HW: Transmit queues on NIC
Apache	HW: Receive queues on NIC
PostgreSQL	App: Application-level spin lock
gmake	App: Serial stages and stragglers
pedsort	HW: Cache capacity
Metis	HW: DRAM throughput

Figure 12: Summary of the current bottlenecks in MOSBENCH, attributed either to hardware (HW) or application structure (App).

5.9 Evaluation summary

Figure 3 summarized the significant scalability improvements resulting from our changes. Figure 12 summarizes the bottlenecks that limit further scalability of MOSBENCH applications. In each case, the application is bottlenecked by either shared hardware resources or application-internal scalability limits. None are limited by Linux-induced bottlenecks.

6 DISCUSSION

The results from the previous section show that the MOSBENCH applications can scale well to 48 cores, with modest changes to the applications and to the Linux kernel. Different applications or more cores are certain to reveal more bottlenecks, just as we encountered bottlenecks at 48 cores that were not important at 24 cores. For example, the costs of thread and process creation seem likely to grow with more cores in the case where parent and child are on different cores. Given our experience scaling Linux to 48 cores, we speculate that fixing bottlenecks in the kernel as the number of cores increases will also require relatively modest changes to the application or to the Linux kernel. Perhaps a more difficult problem is addressing bottlenecks in applications, or ones where application performance is not bottlenecked by CPU cycles, but by some other hardware resource, such as DRAM bandwidth.

Section 5 focused on scalability as a way to increase performance by exploiting more hardware, but it is usually also possible to increase performance by exploiting

a fixed amount of hardware more efficiently. Techniques that a number of recent multicore research operating systems have introduced (such as address ranges, dedicating cores to functions, shared memory for inter-core message passing, assigning data structures carefully to on-chip caches, etc. [11, 15, 53]) could apply equally well to Linux, improving its absolute performance and benefiting certain applications. In future work, we would like to explore such techniques in Linux.

One benefit of using Linux for multicore research is that it comes with many applications and has a large developer community that is continuously improving it. However, there are downsides too. For example, if future processors don't provide high-performance cache coherence, Linux's shared-memory-intensive design may be an impediment to performance.

7 Conclusion

This paper analyzes the scaling behavior of a traditional operating system (Linux 2.6.35-rc5) on a 48-core computer with a set of applications that are designed for parallel execution and use kernel services. We find that we can remove most kernel bottlenecks that the applications stress by modifying the applications or kernel slightly. Except for sloppy counters, most of our changes are applications of standard parallel programming techniques. Although our study has a number of limitations (e.g., real application deployments may be bottlenecked by I/O), the results suggest that traditional kernel designs may be compatible with achieving scalability on multicore computers. The MOSBENCH applications are publicly available at `http://pdos.csail.mit.edu/mosbench/`, so that future work can investigate this hypothesis further.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Brad Chen, for their feedback. This work was partially supported by Quanta Computer and NSF through award numbers 0834415 and 0915164. Silas Boyd-Wickizer is partially supported by a Microsoft Research Fellowship. Yandong Mao is partially supported by a Jacobs Presidential Fellowship. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship.

References

[1] Apache HTTP Server, May 2010. `http://httpd.apache.org/`.

[2] Exim, May 2010. `http://www.exim.org/`.

[3] Memcached, May 2010. `http://memcached.org/`.

[4] PostreSQL, May 2010. `http://www.postgresql.org/`.

[5] The search for fast, scalable counters, May 2010. `http://lwn.net/Articles/170003/`.

[6] J. Aas. Understanding the Linux 2.6.8.1 CPU scheduler, February 2005. `http://josh.trancesoftware.com/linux/`.

[7] AMD, Inc. Six-core AMD opteron processor features. `http://www.amd.com/us/products/server/processors/six-core-opteron/Pages/six-core-opteron-key-architectural-features.aspx`.

[8] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proc. of the 13th SOSP*, pages 95–109, 1991.

[9] J. Appavoo, D. D. Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. Experience distributing objects in an SMMP OS. *ACM Trans. Comput. Syst.*, 25(3):6, 2007.

[10] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[11] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: a new OS architecture for scalable multicore systems. In *Proc of the 22nd SOSP*, Big Sky, MT, USA, Oct 2009.

[12] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, 1990.

[13] D. L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *Computer*, 23(5):35–43, 1990.

[14] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for NUMA memory management. In *Proc. of the 12th SOSP*, pages 19–31, New York, NY, USA, 1989. ACM.

[15] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. D. Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proc. of the 8th OSDI*, December 2008.

[16] R. Bryant, J. Hawkes, J. Steiner, J. Barnes, and J. Higdon. Scaling linux to the extreme. In *Proceedings of the Linux Symposium 2004*, pages 133–148, Ottawa, Ontario, June 2004.

[17] B. Cantrill and J. Bonwick. Real-world concurrency. *Commun. ACM*, 51(11):34–39, 2008.

[18] J. Corbet. The lockless page cache, May 2010. `http://lwn.net/Articles/291826/`.

[19] A. L. Cox and R. J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with platinum. In *Proc. of the 12th SOSP*, pages 32–44, 1989.

[20] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[21] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc of the 22nd SOSP*, Big Sky, MT, USA, Oct 2009.

[22] F. Ellen, Y. Lev, V. Luchango, and M. Moir. SNZI: Scalable nonzero indicators. In *PODC 2007*, Portland, Oregon, USA, Aug. 2007.

[23] GNU Make, May 2010. `http://www.gnu.org/software/make/`.

[24] C. Gough, S. Siddha, and K. Chen. Kernel scalability—expanding the horizon beyond fine grain locks. In *Proceedings of the Linux Symposium 2007*, pages 153–165, Ottawa, Ontario, June 2007.

[25] T. Herbert. rfs: receive flow steering, September 2010. `http://lwn.net/Articles/381955/`.

[26] T. Herbert. rps: receive packet steering, September 2010. `http://lwn.net/Articles/361440/`.

[27] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[28] J. Jackson. Multicore requires OS rework Windows architect advises. *PCWorld magazine*, 2010. `http://www.pcworld.com/businesscenter/article/191914/multicore_requires_os_rework_windows_architect_advises.html`.

[29] Z. Jia, Z. Liang, and Y. Dai. Scalability evaluation and optimization of multi-core SIP proxy server. In *Proc. of the 37th ICPP*, pages 43–50, 2008.

[30] A. R. Karlin, K. Li, M. S. Manasse, and S. S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proc. of the 13th SOSP*, pages 41–55, 1991.

[31] A. Kleen. An NUMA API for Linux, August 2004. `http://www.firstfloor.org/~andi/numa.html`.

[32] A. Kleen. Linux multi-core scalability. In *Proceedings of Linux Kongress*, October 2009.

[33] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proc. of the 21st ISCA*, pages 302–313, 1994.

[34] R. P. LaRowe, Jr., C. S. Ellis, and L. S. Kaplan. The robustness of NUMA memory management. In *Proc. of the 13th SOSP*, pages 137–151, 1991.

[35] J. Li, B. T. Loo, J. M. Hellerstein, M. F. Kaashoek, D. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *Proc. of the 2nd IPTPS*, Berkeley, CA, February 2003.

[36] Linux 2.6.35-rc5 source, July 2010. `Documentation/scheduler/sched-design-CFS.txt`.

[37] Linux kernel mailing list, May 2010. `http://kerneltrap.org/node/8059`.

[38] Y. Mao, R. Morris, and F. Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT, 2010.

[39] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read-copy update. In *Proceedings of the Linux Symposium 2002*, pages 338–367, Ottawa, Ontario, June 2002.

[40] P. E. McKenney, D. Sarma, and M. Soni. Scaling dcache with rcu, Jan. 2004. `http://www.linuxjournal.com/article/7124`.

[41] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[42] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley. Performance issues in parallelized network protocols. In *Proc. of the 1st OSDI*, page 10, Berkeley, CA, USA, 1994. USENIX Association.

[43] D. Patterson. The parallel revolution has started: are you part of the solution or the prolem? In *USENIX ATEC*, 2008. `www.usenix.org/event/usenix08/tech/slides/patterson.pdf`.

[44] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the ACM EuroSys Conference (EuroSys 2010)*, Paris, France, April 2010.

[45] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor system. In *Proceedings of HPCA*. IEEE Computer Society, 2007.

[46] C. Schimmel. *UNIX systems for modern architectures: symmetric multiprocessing and caching for kernel programmers*. Addison-Wesley, 1994.

[47] M. D. Schroeder and M. Burrows. Performance of Firefly RPC. In *Proc. of the 12th SOSP*, pages 83–90, 1989.

[48] J. Stribling, J. Li, I. G. Councill, M. F. Kaashoek, and R. Morris. Overcite: A distributed, cooperative citeseer. In *Proc. of the 3rd NSDI*, San Jose, CA, May 2006.

[49] J. H. Tseng, H. Yu, S. Nagar, N. Dubey, H. Franke, P. Pattnaik, H. Inoue, and T. Nakatani. Performance studies of commercial workloads on a multi-core system. *IEEE Workload Characterization Symposium*, pages 57–65, 2007.

[50] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proc. of the 13th SOSP*, pages 26–40, 1991.

[51] B. Veal and A. Foong. Performance scalability of a multi-core web server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pages 57–66, New York, NY, USA, 2007.

[52] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proc. of the 7th ASPLOS*, pages 279–289, New York, NY, USA, 1996. ACM.

[53] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, 2009.

[54] C. Yan, Y. Chen, and S. Yuanchun. Parallel scalability comparison of commodity operating systems on large scale multi-cores. In *Proceedings of the workshop on the interaction between Operating Systems and Computer Architecture (WIOSCA 2009)*.

[55] C. Yan, Y. Chen, and S. Yuanchun. OSMark: A benchmark suite for understanding parallel scalability of operating systems on large scale multi-cores. In *2009 2nd International Conference on Computer Science and Information Technology*, pages 313–317, 2009.

[56] C. Yan, Y. Chen, and S. Yuanchun. Scaling OLTP applications on commodity multi-core platforms. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 134–143, 2010.

[57] M. Young, A. Tevanian, R. F. Rashid, D. B. Golub, J. L. Eppinger, J. Chew, W. J. Bolosky, D. L. Black, and R. V. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proc. of the 11th SOSP*, pages 63–76, 1987.

Trust and Protection in the Illinois Browser Operating System

Shuo Tang, Haohui Mai, Samuel T. King
University of Illinois at Urbana-Champaign

Abstract

Current web browsers are complex, have enormous trusted computing bases, and provide attackers with easy access to modern computer systems. In this paper we introduce the Illinois Browser Operating System (IBOS), a new operating system and a new browser that reduces the trusted computing base for web browsers. In our architecture we expose browser-level abstractions at the lowest software layer, enabling us to remove almost all traditional OS components and services from our trusted computing base by mapping browser abstractions to hardware abstractions directly. We show that this architecture is flexible enough to enable new browser security policies, can still support traditional applications, and adds little overhead to the overall browsing experience.

1 Introduction

Web-based applications (web apps), browsers, and operating systems have become popular targets for attackers of computer systems. Vulnerabilities in web apps are widespread and increasing. For example, cross-site scripting (XSS), which is effectively a form of script injection into a web app, recently overtook the ubiquitous buffer overflow as the most common security vulnerability [50]. Vulnerabilities in web browsers are less common than web app vulnerabilities, but still occur often. For example, in 2009 Internet Explorer, Chrome, Safari, and Firefox had 349 new security vulnerabilities [4], and attackers exploit browsers commonly [53, 37, 42, 41, 4]. Vulnerabilities in libraries, system services, and operating systems are less common than vulnerabilities in browsers, but are still problematic for modern systems. For example, glibc, GTK+, X, and Linux had 114 new security vulnerabilities in 2009 [1], and in 2009 the most commonly attacked vulnerability was a remote code execution bug in the Windows kernel [4].

However, not all attacks on web apps, browsers, and operating systems are equally virulent. At the top of the computer stack, attacks on web apps, such as XSS, operate within current browser security policies that contain the damage to the vulnerable web app. Moving down the computer stack, attacks on browsers can cause more damage because a successful attack gives the attacker access to browser data for all web apps and access to other resources on the system. At the lowest layers of the computer stack, attacks on libraries, shared system services, and operating systems are the most serious attacks because attackers can access arbitrary states and events, giving them complete control of the system.

Overall, these trends indicate that vulnerabilities higher in the computer stack are more common, but vulnerabilities lower in the computer stack provide attackers with more control and are more damaging. In this paper we focus on preventing and containing attacks on browsers, libraries, system services, and operating systems – the lower layers of the computer stack.

Current research efforts into more secure web browsers help improve the security of browsers, but remain susceptible to attacks on lower layers of the computer stack. The OP web browser [26], Gazelle [52], Chrome [11], and ChromeOS [25] propose new browser architectures for separating the functionality of the browser from security mechanisms and policies. However, these more secure web browsers are all built on top of commodity operating systems and include complex user-mode libraries and shared system services within their trusted computing base (TCB). Even kernel designs with strong isolation between OS components (e.g., microkernels [24, 27, 28] and information-flow kernels [18, 57, 33]) still have OS services that are shared by all applications, which attackers can compromise and still cause damage. Here are a few ways that an attacker can still cause damage to more secure web browsers built on top of traditional OSes:

- A compromised Ethernet driver can send sensitive HTTP data (e.g., passwords or login cookies) to any remote host or change the HTTP response data before routing it to the network stack.
- A compromised storage module can modify or steal any browser related persistent data.
- A compromised network stack can tamper with any network connection or send sensitive HTTP data to an attacker.
- A compromised window manager can draw any content on top of a web page to deploy visual attacks, such as phishing.

In this paper we describe IBOS, an operating system and a browser co-designed to reduce drastically the TCB for web browsers and to simplify browser-based systems. Our key insight is that our lowest-layer software can expose browser-level abstractions, rather than general-purpose OS abstractions, to provide vastly improved security properties for the browser *without* affecting the TCB for traditional applications. Some examples of browser abstractions are cookies for persistent storage, hypertext transfer protocol (HTTP) connections for network I/O, and tabs for displaying web pages. To support traditional applications, we build UNIX-like abstractions on top of our browser abstractions.

IBOS improves on past approaches by removing typically shared OS components and system services from our browser's TCB, including device drivers, network protocol implementations, the storage stack, and window management software. All of these components run above a trusted *reference monitor* [9], which enforces our security policies. These components operate on browser-level abstractions, allowing us to map browser security policies down to the lowest-level hardware directly and to remove drivers and system services from our TCB.

This architecture is a stark contrast to current systems where *all* applications layer application-specific abstractions on top of general-purpose OS abstractions, inheriting the cruft needed to implement and access these general OS abstractions. By exposing application-specific abstractions at the OS layer, we can cut through complex software layers for one particular application without affecting traditional applications adversely, which still run on top of general OS abstractions and still inherit cruft. We choose to illustrate this principle using a web browser because browsers are used widely and have been prone to security failures recently. Our goal is to build a system where a user can visit a trusted web site safely, even one or more of the components on the system have been compromised.

Our contributions are:

- IBOS is the first system to improve browser and OS security by making browser-level abstractions first-class OS abstractions, providing a clean separation between browser functionality and browser security.
- We show that having low-layer software expose browser abstractions enables us to remove almost all traditional OS components from our TCB, including device drivers and shared OS services, allowing IBOS to withstand a wide range of attacks.
- We demonstrate that IBOS can still support traditional applications that interact with the browser and shared OS services without compromising the security of our system.

2 The IBOS architecture

This paper presents the design and implementation of the IBOS operating system and browser that reduce the TCB for browsing drastically. Our primary goals are to enforce today's browser security policies with a small TCB, without restricting functionality, and without slowing down performance. To withstand attacks, IBOS must ensure any compromised component (1) cannot tamper with data it should not have access to, (2) cannot leak sensitive information to third parties, and (3) cannot access components operating on behalf of different web sites.

In this section we discuss the design principles that guide our design and the overall system architecture. In Section 4 we discuss the security policies and mechanisms we use.

2.1 Design principles

We embrace microkernel [27], Exokernel [19], and safety kernel design principles in our overall architecture. By combining these principles with our insight about exposing browser abstractions at the lowest software layer we hope to converge on a more trustworthy browser design. Five key principles guide our design:

1. *Make security decisions at the lowest layer of software.* By pushing our security decisions to the lowest layers we hope to avoid including the millions of lines of library and OS code in our TCB.
2. *Use controlled sharing between web apps and traditional apps.* Sharing data between web apps and traditional apps is a fundamental functionality of today's practical systems and should be supported. However, this sharing should be facilitated through a narrow interface to prevent misuse.

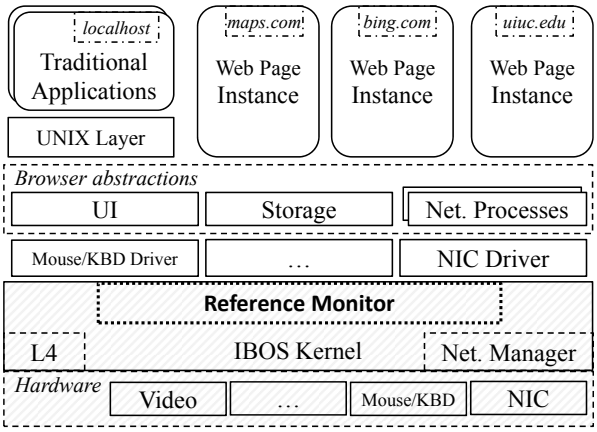


Figure 1: Overall IBOS architecture. Our system contains user-mode drivers, browsers API managers, web page instances, and traditional processes. To manage the interactions between these components, we use a reference monitor that runs within our IBOS kernel. Shaded regions make up the TCB.

3. *Maintain compatibility with current browser security policies.* Our primary goal is to improve the enforcement of current browser policies without changing current web-based applications.

4. *Expose enough browser states and events to enable new browser security policies.* In addition to enforcing current browser policies, we would like our architecture to adapt easily to future browser policies.

5. *Avoid rule-based OS sandboxing for browser components.* Fundamentally, rule-based OS sandboxing is about restricting unused or overly permissive interfaces exposed by today's operating systems. However, sandboxing systems can be complex (the Ubuntu 10.04 SELinux reference policy uses over 104K lines of policy code) and difficult to implement correctly [23, 51]. If our architecture requires OS sandboxing for browser components then we should rethink the architecture.

2.2 Overall architecture

Figure 1 shows the overall IBOS architecture. The IBOS architecture uses a basic microkernel approach with a thin kernel for managing hardware and facilitating message passing between processes. The system includes user-mode device drivers for interacting directly with hardware devices, such as network interface cards (NIC), and browser API managers for accessing the drivers and implementing browser abstractions. The key browser abstractions that the browser API managers implement are HTTP requests, cookies and local storage for storing persistent data, and tabs for displaying user-interface (UI) content. Web apps use these abstractions directly to implement browser functionality, and traditional applications (traditional apps) use a UNIX layer to access UNIX-like abstractions on top of these browser abstractions.

2.2.1 The IBOS kernel

Our IBOS kernel is the software TCB for the browser and includes resource management functionality and a reference monitor for security enforcement. The IBOS kernel also handles many traditional OS tasks such as managing global resources, creating new processes, and managing memory for applications. To facilitate message passing, the IBOS kernel includes the L4Ka::Pistachio [8] message passing implementation and MMU management functions. All messages pass through our reference monitor and are subjected to our overall system security policy. Section 4 describes the policies that the IBOS kernel enforces and the mechanisms it uses to implement these policies.

2.2.2 Network, storage, and UI managers

The IBOS network subsystem handles HTTP requests and socket calls for applications. To handle HTTP requests, *network processes* check a local cache to see if the request can be serviced via the cache, fetch any cookies needed for the request, format the HTTP data into a TCP stream, and transform that TCP stream into a series of Ethernet frames that are sent to the NIC driver. *Socket network processes* export a basic socket API and simply transform TCP streams to Ethernet frames for transmission across the network. Only traditional apps can access our socket network processes. The IBOS kernel manages global states, like port allocation.

The IBOS storage manager maintains persistent storage for key-value data pairs. The browser uses the storage manager to store HTTP cookies and HTML5 local storage objects, and the basic object store includes optional parameters, such as `Path` and `Max-Age`, to expose cookie properties to the reference monitor. The storage manager uses several different namespaces to isolate objects from each other. Web apps and network processes share a namespace based on the origin (the `<protocol, domain name, port>` tuple of a uniform resource locator) that they originate from, and web apps and traditional apps share a "localhost" namespace, which is separate from the HTTP namespace. All other drivers and managers have their own pri-

vate namespaces to access persistent data.

The IBOS UI manager plays the role of the window manager for the system. However, rather than implement the browser UI components on top of the traditional window motif, we opted for a tabbed browser motif. Basic browser UI widgets, called the browser chrome, are displayed at the top of the screen. IBOS displays web pages in tabs and the user can have any number of tabs open for web apps. There is a tab for basic browser configuration and administration, and a tab that is shared by traditional apps. If traditional apps wish to implement the window motif, they can do so within the tab. The main advantage of our browser-based motif is that it enables IBOS to bypass the extra layers of indirection traditional window managers put between applications and the underlying graphics hardware, exposing browser UI elements and events directly to the IBOS kernel. We discuss the security implications of our design decision in more detail in Section 4.8.

2.2.3 Web apps, traditional apps, and plugins

The IBOS system supports two different types of processes: web page instances and traditional processes. A web page instance is a process that is created for each individual web page a user visits. Each time the user clicks on a link or types a uniform resource locator (URL) into the address bar, the IBOS kernel creates a new web page instance. Web page instances are responsible for issuing HTTP requests, parsing HTML, executing JavaScript, and rendering web content to a tab. Traditional processes can execute arbitrary instructions, and the key difference between a web page instance and a traditional processes is that the IBOS kernel gives them different security labels, which the kernel uses for access control decisions. Web page instances are labeled with the origin of the HTTP request used to initiate the new web page, and traditional processes are labeled as being from "localhost." These two processes interact via the storage subsystem since both types of processes can access "localhost" data.

In general, plugins are external applications that browsers use to render non-HTML content. One common example of a plugin is the Flash player that enables browsers to play Flash content. In IBOS, plugins run as traditional processes, except that they are launched by the browser and the system gives them access to browser states and events through a standard plugin programming interface, called the NPAPI [2].

3 Current browser policies

In this section we give a brief introduction to the same-origin policy (SOP) for browser security. For a more complete discussion of this policy and others, plus experimental results showing how current browsers implement them, please see a recent paper by Singh, *et al.* [47].

The primary security policy that all modern browsers implement is the SOP. The SOP acts as a non-interference policy for the web. Loosely speaking, the SOP provides isolation for web pages and states that come from different origins – origins are used as labels for browser access control policies. If the browser has a web page open from `uiuc.edu` and from `attacker.com`, the SOP should ensure that these two web pages are isolated from each other. Unfortunately, Chrome, IE8, Safari, and Firefox all enforce the SOP using a number of checks scattered throughout the millions of lines of browser code and current browsers have had trouble implementing the SOP correctly [14].

In a browser, a frame is a container that encapsulates a HTML document and any material included in that HTML document. Web pages are frames, and web developers can embed additional frames within web pages – these frames are called `iframes`. Developers can include `iframes` from the same origin as the hosting frame, or from a different origin. Each frame is labeled with the origin of the main HTML document used to populate the frame, meaning that a cross-origin `iframe` has a different label than the hosting web page.

In general HTML documents include references to network objects that the browser will download and display to form the web page. These network objects can be images, JavaScript, and CSS. Browsers can download these objects from any domain and the browser labels them with the origin of the hosting frame. For example, if a page from `uiuc.edu` includes a script from `foo.com`, that script runs with full `uiuc.edu` permissions and can access any of the states in that web page. Browsers can also download HTML documents and XML HTTP requests (used for Ajax), but the SOP dictates that these objects must come from the same origin as the hosting frame.

4 IBOS security policies and mechanisms

Our primary goal is to enforce browser security policies from within our IBOS kernel. This section describes the mechanisms that the IBOS kernel uses to enforce the SOP. We also discuss policies and mechanisms for enforcing UI interactions, and we describe a custom policy engine that lets web sites further restrict current policies.

4.1 Threat model and assumptions

Our primary goal is to ensure that the IBOS kernel upholds our security policies even if one or more of the subsystems have been compromised. In our threat model,

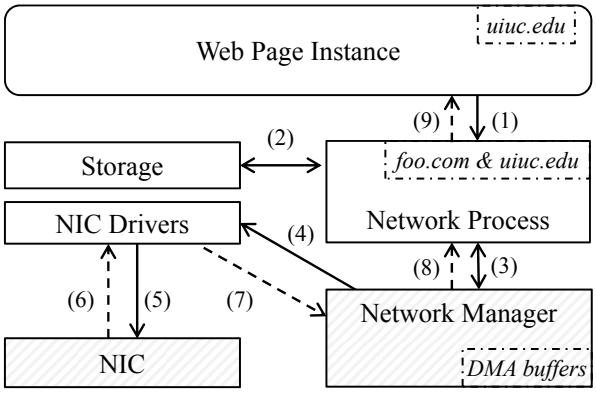


Figure 2: This figure enlarges the right half of Figure 1 and shows how our IBOS subsystems interact when a web page instance from `uiuc.edu` issues a network request to `foo.com`. Subsystems are shown in boxes and solid and dotted arrows represent IBOS messages for outgoing and incoming data respectively. The reference monitor (which is not shown here) checks all these messages to enforce security properties.

we assume that an attacker controls a web site and can serve arbitrary data to our browser, or that the system contains a malicious traditional app. We also assume that this malicious data or traditional app can compromise one or more of the components in our system. These susceptible components include all drivers, browser API managers, web page instances, and traditional processes. Once the attacker takes control of these components, we assume that he or she can execute arbitrary instructions as a result of the attack. We focus on maintaining the integrity and confidentiality of the data in our browser. In other words, we would like the user to be able to open a web page on a trusted web server, and interact with this web page securely, even if everything on the client system outside of our TCB has been compromised. Availability is an important, but separate, aspect of browser security that we do not address in this paper.

In our system we trust the layers upon which we built IBOS. These layers include the IBOS kernel and the underlying hardware. Like all other browsers, IBOS predicates security decisions based on domain names, so we trust domain name servers to map domain names to IP addresses correctly. Compromising any of these trusted layers compromises the security of IBOS.

4.2 IBOS work flow

This section describes a web page instance making a network request to help illustrate the security mechanisms that IBOS uses.

Figure 2 shows the flow of how a web page instance fetches data from the network. The user visits a page hosted at `uiuc.edu` and this web page includes an image from `foo.com`. To download the image, (1) the web page instance will make an HTTP request that the IBOS kernel forwards to an appropriate network process. The network process forms a HTTP request, which includes setting up HTTP headers, (2) fetching cookies from the storage subsystem, (3) requesting a free local TCP port to transform this request into TCP/IP packets and Ethernet frames, and (4) sending it to network manager. The network manager notifies the Ethernet driver which (5) programs the NIC to transmits the packet out to the network. When the NIC receives a reply for the request, (6) it notifies the Ethernet driver. The driver subsequently (7) notifies the network manager, which (8) forwards the packet to the appropriate network process. The network process then parses the data and (9) passes the resulting HTTP reply and data to the original web page instance.

4.3 IBOS labels

To enforce access control decisions, the IBOS kernel labels web page instances, traditional processes, and network processes. IBOS labels specify the resources that a process can access or messages it can receive. Each web page instance has one label, which is the origin of the main HTML document. Each traditional process is labeled as being from "localhost" when they are created. Each network process has an origin label for the network resources it handles and has an origin label for the web page instances that are allowed to access it. IBOS labels the processes upon creation, and keeps the labels unchanged throughout the processes' life-cycle.

An important point is that the IBOS kernel infers the origin labels for web page instances and network processes automatically by extracting related information from the messages passed among them. By inferring labels rather than relying on processes to label themselves, the IBOS kernel ensures that it has the correct label information, even if a process is compromised.

The *newUrl* and *fetchUrl* IBOS system calls are the two requests that cause the kernel to label processes. The *newUrl* system call is used by web page instances and the UI manager use to navigate the browser to a new URL. The *newUrl* system call consists of two arguments: a URL and a byte array for HTTP POST data. When the IBOS kernel receives a *newUrl* request it will create a new web page instance and set the label for this web page instance by parsing the origin out of the URL argument of the *newUrl* request. When servicing *newUrl* requests, the IBOS kernel will reuse old web page instances (to reduce process startup times), but only when the origin labels match for the old web page instance and the URL

argument.

Web page instances use the *fetchUrl* system call to issue HTTP and HTTPS requests to fetch network objects, such as images. The *fetchUrl* system call has two arguments: a URL and HTTP header information. When a web page instance issues a *fetchUrl* system call, the IBOS kernel uses the origin of the web page instance (set by the original *newUrl* call) and the origin of the *fetchUrl* URL argument to find a network process with these same labels, or creates a new network processes and labels it accordingly if an existing network process cannot be found.

More details about how we use these labels for access control decisions are described in the remainder of this section.

4.4 Security invariants

For all of our subsystems, we use *security invariants* that are assertions on all interactions between subsystems that check basic security properties. The key to our security invariants is that we can extract security relevant information from messages automatically, and provide high assurance that the system maintains the security policy without having to understand how each individual subsystem is implemented. Using these security invariants, we remove from the TCB almost all of the components found in modern commodity operating systems, including device drivers.

The ideal security invariant is complete, implementation agnostic, executes quickly, and requires only a small amount of code in the IBOS kernel. A complete invariant can infer all of the states needed to ensure the high-level security policy, and an implementation agnostic invariant can infer states without relying on the specific implementation of individual subsystems. The IBOS kernel evaluates invariants in the kernel and inline with messages, so security invariants should execute quickly and require little code to implement. In our design we strive to make the appropriate trade offs among these properties to improve security without making the system slow or increasing our TCB significantly. The base security invariant we have is:

SI 0: *All components can only perform their designated functions.*

For example, the UI subsystem can never ask for cookie data or the storage manager cannot impersonate a network process to send synthesized attack HTTP data to a web page instance.

4.5 Driver invariants

The two driver invariants the IBOS kernel enforces are:

SI 1: *Drivers cannot access DMA buffers directly.*
SI 2: *Devices can only access validated DMA buffers.*

In our approach, we use a split driver architecture where we separate the management of device control registers from the use of device buffers (SI 1). For example, our Ethernet driver never has access to transmit or receive buffers directly. Instead, it knows the physical addresses where the IBOS kernel stores these buffers, and it programs the NIC to use them. By separating these two functions we can interpose on the communications between them to ensure that IBOS upholds browser security policies, even if an attacker completely compromises a shared driver.

Using this split architecture, processes fill in device-specific buffers for DMA transfers, and the IBOS kernel infers when drivers initiate DMA transfers to ensure that the driver instructs the device to use a verified DMA buffer (SI 2). Fortunately, DMA buffers tend to use well-defined interfaces, like Ethernet frames for Ethernet drivers, so the IBOS kernel can readily glean security relevant information from these DMA buffers before the device accesses them. Unfortunately, the interface between drivers and devices is device-specific, so the IBOS kernel must have a small state machine for each device to properly infer DMA transfers. However, we found this state machine to be quite small for the devices that we use in IBOS.

In IBOS we implement a driver for the e1000 NIC, a VESA BIOS Extensions driver for our video card, and drivers for the mouse and keyboard.

4.6 Storage invariants

The primary invariant we strive to enforce in the storage manager is:

SI 3: *All of our key-value pairs maintain confidentiality and integrity even if the storage stack itself becomes compromised.*

To enforce this invariant, our IBOS kernel encrypts all objects before passing them to the storage subsystem. To encrypt data, the IBOS kernel maintains separate encryption keys for all of the namespaces on the IBOS system. These namespaces include separate namespaces for HTTP cookies based on the domain of the cookie, separate namespaces for web page instances based on the origin of the page, separate namespaces for each of our subsystems, and a separate namespace for all traditional apps. When the IBOS kernel passes a request to the storage manager it will append the security labels, a copy of the key from the key-value pair, and a hash of the contents to the payload before encrypting the data and

passing it to the storage subsystem. When the IBOS kernel retrieves this data, it can decrypt the data and check the labels and integrity of the information. By using encryption, the IBOS kernel does not need to implement security invariants for any of our storage drivers, and our storage subsystem is free to make data persistent using any mechanisms it sees fit, such as the network (like in our implementation) or via a disk-based storage system.

Our current implementation does not make any efforts to avoid an attacker that deletes objects or replays old storage data. For web applications this limitation has only a small effect because the cookie standards do *not* require browsers to keep cookies persistently and because web applications often limit the lifetime of cookies using expiration dates, which are also part of the cookie standard. However, if this limitation did become problematic, we could apply the principles learned from distributed or secure file systems to provide stronger guarantees.

4.7 Network process invariants

Our IBOS kernel maintains five main invariants for network processes:

SI 4: *The kernel must route network requests from web page instances to the proper network process.*
SI 5: *The kernel must route Ethernet frames from the NIC to the proper network processes.*
SI 6: *Ethernet frames from network processes to the NIC must have an IP address and TCP port that matches the origin of the network process.*
SI 7: *HTTP data from network processes to web page instances must adhere to the SOP.*
SI 8: *Network processes for different web page instances must remain isolated.*

To help enforce these invariants, IBOS puts all network processes in their own protection domains. If a web page instance makes a HTTP request, the kernel will extract the origin from the request message and either route this request to an existing network process that has the same label, or it will create a new network process and label the network process with the origin of the HTTP request. Likewise, the kernel inspects incoming Ethernet frames to extract the origin and TCP port information, and routes these frames to the appropriately labeled network process. By putting network processes in their own protection domains, the kernel naturally ensures that network requests from web page instances and Ethernet frames from the NIC are routed to the correct network process (SI 4) (SI 5).

To ensure that the NIC sends outgoing Ethernet frames to the correct host, the IBOS kernel checks all outgoing Ethernet frames before sending them to the NIC to check the IP address and TCP port against the label of the sending network process (SI 6). Also, the IBOS kernel checks cookies before passing them to the network process to ensure that all of the origin labels adhere to cookie standards. By performing these checks, the IBOS kernel ensures that the NIC sends outgoing network requests to the proper host and that the request can only include data that would be available to the server anyway.

To enforce the SOP, the IBOS kernel inspects HTTP data before forwarding it to the appropriate web page instance and drops any HTML documents from different origins (SI 7). To inspect data, the kernel uses the content sniffing algorithm from Chrome [10] to identify HTML documents so the kernel can check to make sure that the origin of HTML documents and the origin of the web page instance match. This countermeasure prevents compromised web page instances from peering into the contents of a cross-origin HTML document, thus preventing the compromised web page instance from reading sensitive information included in the HTML document.

To help isolate web page instances from each other, we also label network processes with the origin of the web page instance (SI 8). This second label is used only for network access control decisions and does *not* affect the cookie policy, which is predicated on the origin of the network request. To access network processes, the origin of the web page instance must match the origin of this second label. By using this second label, the IBOS kernel isolates network requests from different web page instances to the same origin. As a result of this isolation, a web page instance that is served a malicious network resource (e.g., a malicious ad [41]) that compromises a network process remains isolated from other web page instances. If an attacker can compromise a network process, IBOS limits the damage to the web page instance that included the malicious content.

4.8 UI invariants

The three UI invariants that the IBOS kernel enforces are:

SI 9: *The browser chrome and web page content displays are isolated.*
SI 10: *Only the current tab can access the screen, mouse, and keyboard.*
SI 11: *The URL of the current tab is displayed to the user.*

The key mechanisms that our UI subsystem uses to provide isolation are to use a frame buffer video driver and page protections to isolate portions of the screen (SI 9). Our video driver uses a section of memory, called a frame buffer, for writing to the screen. Processes

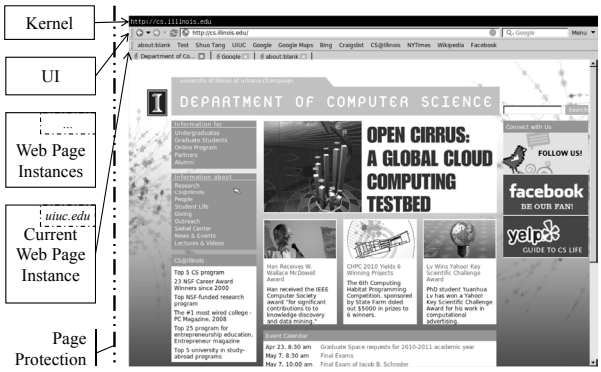


Figure 3: IBOS display isolation. This figure shows how IBOS divides the display into three main parts: a bar at the top for the kernel, a bar for browser chrome, and the rest for displaying web page content. The IBOS kernel enforces this isolation using page protections and *without* relying on a window manager.

write pixel values to this frame buffer and the graphics card displays these pixels. Although our mechanism makes heavy use of the software rastering available in Qt Framework[3], our experiences and anecdotal evidence from the Qt developers shows that software rastering can perform roughly as fast as native X drivers running on Linux [7]. The key advantage of our approach is that the IBOS kernel can use standard page-protection mechanisms to isolate portions of the screen. Although our current implementation does not support hardware acceleration, we believe that our techniques will work because the IBOS kernel can interpose on standardized acceleration hardware/software interfaces, such as OpenGL and DirectX.

To provide screen isolation, we divide up the screen into three horizontal portions (Figure 3). At the top, we reserve a small bar that only the IBOS kernel can access. We use the next section of the screen for the UI subsystem to draw the browser chrome. Finally, we provide the remainder of the screen to the web page instance. To ensure that only one web page instance can write to the screen at any given time, we only map the frame buffer memory region into the currently active web page instance and we only route mouse and keyboard events to this currently active web page instance (SI 10).

To switch tabs, the UI subsystem notifies the IBOS kernel about which tab is the current tab, and the IBOS kernel updates the frame buffer page table entries appropriately. However, a malicious UI manager could switch tabs arbitrarily and cause the address bar and the tab content to become out of sync (e.g., shows a page from `attacker.com`, but claims the page comes from `uiuc.edu`). One alternative we considered for this UI inconsistency was interposing on mouse and keyboard clicks to infer which tab the user clicked on, and also performing optical character recognition on the address bar to determine the address that the UI manager is displaying. However, tracking this level of detail would require far too much implementation specific information and would require the IBOS kernel to track additional events like a user switching the order of tabs.

Our approach for the IBOS kernel is to use the kernel display area to display the URL for the currently visible web page instance (SI 11). The kernel derives the URL from the label of the currently visible web page instance, providing high assurance that the URL the kernel displays matches the URL of the visible web page instance without tracking implementation specific states and events in the UI manager. Although this security invariant appears simple, it is something that modern web browsers have had trouble getting right [13].

4.9 Web page instances and `iframes`

The IBOS kernel creates a new web page instance each time a user clicks on a link or types a new URL in the address bar. To enforce the SOP on `iframes`, we run cross-origin `iframes` in separate web page instances. This separation allows us to fully track the SOP using kernel visible entities. To facilitate communication between web page instances and the `iframes` that they host, we marshal `postMessage` calls between the two.

Our current display isolation primitives are coarse grained and we rely on the web page instance to manage cross-origin `iframe` displays even though `iframes` run in separate protection domains. However, current display policies allow web page instances to draw over cross-origin `iframes` that they host, so this design decision has no impact on current browser policies. One potential shortcoming of this display management approach is that compromised web page instances can read the display data for embedded `iframes`. Fortunately, many sites with sensitive information, like `facebook.com` and `gmail.com`, use frame busting techniques [34] to prevent cross-origin sites from embedding them, which the IBOS kernel can enforce.

4.10 Custom policies

Our main focus of this project is being able to enforce current browser policies from the lowest layer of software. However, we also want to create an architecture that exposes enough browser states and events to enable novel browser security policies. Attacks such as XSS operate within traditional browser policies and can be difficult to prevent without relying on the HTML or JavaScript engine implementations. Although our archi-

tecture cannot prevent XSS, our goal is to prevent these types of attacks from causing damage.

One mechanism we implement in IBOS is to give a web server the ability to create its own more restrictive security policy to prevent attacks from sending sensitive information to third-party hosts. In our custom policy, we allow web sites to specify a server-side policy file that IBOS retrieves to restrict network accesses for a web page instance, similar to Tahoma manifests [15]. For example, assume that a bank website located at `http://www.bank.com` creates a policy file at `http://www.bank.com/.policy` that specifies the online bank system can only access resources from `www.bank.com` or `data.bank.com`. IBOS retrieves the policy file and automatically applies a more restrictive policy for the online bank web application. This restrictive policy prevents an attacker from sending stolen information to a third-party host, providing an additional layer of protection for the web application.

5 Implementation

The implementation of IBOS is divided into three parts: the IBOS kernel, IBOS messaging passing interfaces, and IBOS subsystems. The IBOS kernel is implemented on top of the L4Ka::Pistachio microkernel and runs on X86-64 uniprocessor and SMP platforms. We modified L4Ka to improve its support for SMP systems. The IBOS kernel schedules processes based on a static priority scheduling algorithm.

The IBOS kernel provides three basic APIs (i.e., `send()`, `recv()`, and `poll()`) to facilitate message passing. Applications use `send()` and `recv()` for communication and call `poll()` to wait for new messages. The IBOS kernel intercepts all messages and automatically extracts the semantics from them, like creating a new web page instance or forwarding cookies to network processes. Then the kernel inspects the semantics to make sure they conform to all security invariants and policies that we described in previous sections.

The IBOS subsystems implements APIs for web browsers and traditional applications. They are built on top of an IBOS-specific uClibc [6] C library, lwIP [17] TCP/IP stack and the Qt Framework [3]. The web browser also uses an IBOS-specific WebKit [5] to parse and render web pages.

To support traditional apps, we use our uClibc and Qt implementations to provide access to browser abstractions using the UNIX-like abstractions of the C runtime, and GUI support from Qt. We use a few Qt sample programs for testing and we implement one plugin. Our plugin is a PDF viewer that uses the Ghostscript PDF rendering engine with bindings for Qt.

System	LOC
IBOS	**42,044**
IBOS Kernel	8,905
L4Ka::Pistachio	33,139
Firefox on Linux	**> 5,684,639**
Firefox 3.5	2,171,267
GTK+ 2.18	489,502
glibc 2.11	740,314
X.Org 7.5	653,276
Linux kernel 2.6.31	1,630,280
ChromeOS	**> 4,407,066**
Chrome browser kernel 4.1.249	714,348
GTK+ 2.18	489,502
glibc 2.11	740,314
ChromeOS kernel & services (May 2010)	2,462,902

Table 1: Estimation of LOC of TCBs for IBOS, Firefox on Linux, and ChromeOS. LOC counts are also shown for some major components that are included in the TCB.

6 Evaluation

This section describes our evaluation of IBOS. In our evaluation, we analyze the security of IBOS by measuring the number of lines of code (LOC) in the IBOS TCB and comparing it with other systems, and by looking at recent bugs in comparable systems and counting vulnerabilities that IBOS is susceptible to. We also revisit the example attacks we discussed in the introduction, and we measure the performance.

6.1 TCB

In IBOS, our goal is to minimize the TCB for web browsers and to simplify browser-based systems. To quantitatively evaluate our effort, we count the LOC in the IBOS TCB and compare it against the TCB for Firefox and ChromeOS. IBOS supports fewer hardware architectures, platforms, device drivers and features, such as browser extensions, than Firefox running on Linux and ChromeOS. For a fair comparison, we only count source code that is used for running above Linux and on the X86-64 platform. Also, we omit all device drivers from our counts except for the drivers we implement in IBOS.

Table 1 shows the result of LOC counts in the TCB for these three systems, measured by SLOCCount [54]. For Firefox and ChromeOS, our counts are conservative because we only count the major components that make up the TCB for each system – there are likely more component that are also in the TCB for these systems. Because the IBOS TCB has only around 42K LOC, it is possible to formally verify or manually review the entire IBOS TCB. And in fact, one L4 type microkernel has already

Affected Component	Num.	Prevented
Linux kernel overall	21	20 (95%)
File system	*12*	*12 (100%)*
Network stack	*5*	*5 (100%)*
Other	*4*	*3 (75%)*
X Server	2	2 (100%)
GTK+ & glibc	5	5 (100%)
Overall	28	27 (96 %)

Table 2: OS and library vulnerabilities. This table shows the number of vulnerabilities that IBOS prevents.

been formally verified [32].

6.2 OS and library vulnerabilities

To evaluate the security impact of IBOS's reduced TCB, we obtained a list of 74 vulnerabilities found in the Linux kernel, X Server, GTK+, and glibc this year so far (as of Sep. 18, 2010) [1] to see how the IBOS architecture handles them. Out of the 74 vulnerabilities, 20 are related to unsupported hardware architectures and devices, and 26 cause denial-of-service, which is out-of-scope for this paper. For the remaining 28, we classify them based on the subsystem the vulnerability lies in to determine if IBOS is susceptible to these vulnerabilities.

Table 2 shows IBOS is able to prevent 27 of 28 vulnerabilities (96%). The only vulnerability we miss is a memory corruption vulnerability in the e1000 Ethernet driver. Normally IBOS is *not* susceptible to bugs in device drivers, but this particular bug resulted from the driver not accounting properly for Ethernet frames larger than 1500 bytes, and this type of logic is what our NIC verification state machine uses, so we counted this bug against IBOS.

6.3 Browser vulnerabilities

To evaluate security improvements that IBOS makes for browsers themselves, we compared how well IBOS could contain or prevent vulnerabilities found in Google's Chrome browser. For this evaluation, we obtained a list of 295 publicly visible bugs with the "security" label in Chrome's bug tracker. Out of the 295 bugs, 42 cause denial-of-service such as a simple crash or 100% CPU utilization. IBOS does not address denial-of-service or resource management currently. An additional 78 are either invalid, duplicate, not actually security issues, or related to features that IBOS does not have, such as browser extensions. For the remaining 175 bugs, we examined each of them to the best of our knowledge and classified them into the following seven categories and compared how Chrome and IBOS handle those cases:

Memory exploitation: an attacker could use a memory corruption bug to deploy a remote code execution attack. For Chrome, if the bug is in its rendering engine, Chrome contains the attack. However, bugs in the browser kernel give attackers access to the entire browser. For IBOS, bugs in either the rendering engine or other service components are contained as they are all out of the TCB.

XSS: browsers rely on careful sanitization and correct processing of different encodings to prevent XSS attacks. For both Chrome and IBOS, it is infeasible to eliminate XSS attacks, but they both contain the attacks in the affected web apps.

SOP circumvention: Chrome runs contents in frames from different origins in a single address space and uses scattered "if" and "else" statements to enforce the same-origin policy. This logic can be sometime subverted. In IBOS, we run iframes in different web page instances to provide strong isolation and check cross-origin access in the IBOS kernel.

Sandbox bypassing: Chrome uses sandboxing techniques, such as SELinux, to limit the rendering engine's authority. However, rule-based sandboxing is complex and can be bypassed in some scenarios. In IBOS, we designed browser abstractions to restrict the authority of each subsystem, which are immune to this kind of problem naturally.

Interface spoofing: browsers are sometime vulnerable to visual attacks in which a malicious website can use complex HTTP redirection or even replicate the "look and feel" of victim websites to deploy phishing. Chrome uses a blacklist-based filter to warn users of malicious websites. In IBOS, the IBOS kernel separates the display of different web page instances and uses the labels of web page instances to display the correct URL in the top of the screen to give the user a visual cue of which website he or she is visiting.

UI design flaw: some security concerns arise because of careless implementation, such as showing users' passwords in plain text. Both Chrome and IBOS are vulnerable to this type of problem.

Misc: some vulnerabilities could not easily be classified and mostly have low security severity. This is the category for those remaining bugs.

In Table 3, we show the detailed results of the analysis of the 175 vulnerabilities, broken down by the classifications above. We examined each of them to determine whether Chrome contains the threats in the affected components, and whether IBOS contains or eliminates the attacks. The table shows IBOS successfully protects users from 135 of the 175 vulnerabilities (77%).

The largest portion of bugs are browser implementation flaws that cause memory corruption and allow remote code execution. Chrome does a fairly good job containing most of them when they are in the rendering

			Chrome	IBOS
Category	**Example**	**Num.**	**Contained**	**Contained or eliminated**
Memory exploitation	A bug in layout engine leads to remote code execution	82	71 (86%)	79 (96%)
XSS	XSS issue due to the lack of support for ISO-2022-KR	14	12 (87%)	14 (100%)
SOP circumvention	XMLHttpRequest allows loading from another origin	21	0 (0%)	21 (100%)
Sandbox bypassing	Sandbox bypassing due to directory traversal	12	0 (0%)	12 (100%)
Interface spoofing	Two pages merge together in certain situation	6	0 (0%)	6 (100%)
UI design flaw	Plain-text information leak due to autosuggest	17	0 (0%)	0 (0%)
Misc	Geolocation events fire after document deletion	22	0 (0%)	3 (14%)
Overall		175	83 (46%)	135 (77%)

Table 3: Browser vulnerabilities. This table shows the number of Chrome vulnerabilities that Chrome itself contains and IBOS contains or eliminates.

engine. However, Chrome is unable to contain exploits in the browser kernel. A good example is a bug in the HTTP chunked encoding module in the browser kernel, which opens the possibility for a remote attacker to inject code. In IBOS, the TCP/IP and HTTP stack is pushed out of the TCB, and is replicated and isolated according to browser security policies. Thus, IBOS is able to contain this bug. The three memory corruption bugs IBOS could not contain were from bugs in Chrome's message passing system. Because the IBOS message passing logic resides within our TCB, we counted these bugs as bugs that IBOS would have missed.

6.4 Motivation revisited

In the introduction, we listed some examples of attacks that an attacker can use to still cause damage to modern secure web browsers by exploiting code in their TCB. We revisit these examples again to argue that IBOS can prevent them.

A compromised Ethernet driver cannot access the DMA buffers used by the device. Even if an attacker exploits the Ethernet driver, he or she still cannot tamper with network packets because the driver does not have access to DMA buffers and because the IBOS kernel validates all transmit and receive buffers that the driver sets.

A compromised storage module has little impact on data confidentiality and integrity. The IBOS kernel encrypts all data with secret keys that only the IBOS kernel has access to. Stored objects are tagged with a hash and origin information so that the IBOS kernel is able to detect tampered data. The only thing a compromised storage module can do is delete objects.

A compromised network stack is constrained as well. In IBOS, every network process runs a complete network stack. A compromised network process cannot send users' data to a third party host as the IBOS kernel ensures it can only communicate with the expected host. Network processes do have the ability to modify or replay HTTP requests, but the web server might have a mechanism to defend against replay attacks.

A Compromised window manager cannot affect other subsystems in IBOS. In IBOS, the role of window manager is simplified to only draw the browser chrome. It can change some potentially sensitive information, such web page titles. However, the IBOS kernel displays the URL of the current tab in the kernel display area, providing users with some visual cues as to the provenance of the displayed web content.

6.5 Performance

To evaluate the performance implication of IBOS's architecture, we compare its browsing experience to other web browsers running in Linux. All experiments were carried out on a 2.33GHz Intel Core 2 Quad CPU Q8200 with 4GB of memory, a 320GB 7200RPM Seagate ST3320613 SATA hard drive and an Intel PRO/1000 NIC connected to 1000 Mbps Ethernet. For Linux, we used Ubuntu 9.10 with kernel version 2.6.31-16-generic (x86-64).

We use page load latency to represent browsing experience. Page load latency is defined as the elapsed time between initial URL request and the DOM `onload` event. We compare IBOS with Firefox 3.5.9, Chrome for Linux 4.1.249. We also ported most of the IBOS browser components to Linux platform (noted as IBOS-Linux) to focus on the performance impact of our IBOS kernel architecture. In IBOS, we statically allocate processors for subsystems as follows: the kernel and device drivers run on CPU0, network processes run on CPU1, web page instances run on CPU2, and all other components run on CPU3. IBOS, IBOS-Linux, and Chrome all use a same version of WebKit from February 2010 with just-in-time JavaScript compilation and HTTP pipelining enabled. For the WebKit-based browsers, we instrument them to measure the time in between the initial URL request and the DOM `onload` event. For Firefox, we use an extension that measures these same events. To reduce noise introduced by our network connection, we load

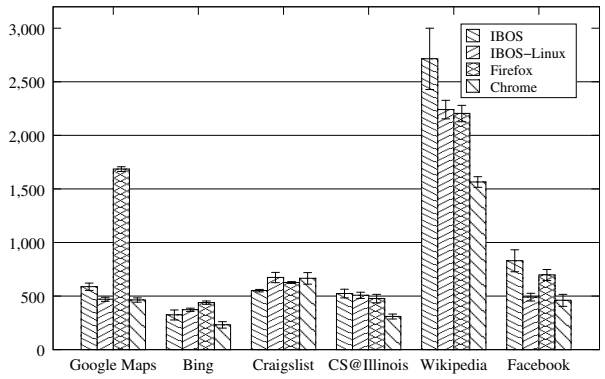


Figure 4: Page load latencies for IBOS and other web browsers. All latencies are shown in milliseconds.

each web site using a fresh web page/browser instance with an empty cache 15 times and report the average of the five shortest page load latency times.

In Figure 4, we present the page load latency times for six popular websites and show the standard deviations with the error bars. Overall, Chrome has the shortest page load latencies due to its effective optimization techniques. For `maps.google.com`, IBOS, IBOS-Linux, and Chrome out-perform Firefox, possibly due to optimization in the WebKit engine for this particular site. For `www.bing.com`, `sfbay.craigslist.org` and `cs.illinois.edu`, IBOS, IBOS-Linux, and Firefox show roughly the same results. IBOS has the fastest loading time for `craigslist`. `Craigslist` is a simple web site with few HTTP requests and with a large number of HTML elements. We hypothesize that the small performance improvement is due to the simplified IBOS software stack.

Both `en.wikipedia.org/wiki/Main_Page` and `www.facebook.com` have more HTTP requests than any of the other sites, and we observe slower page load latencies for IBOS than for other browsers. For these experiments IBOS performs slower than IBOS-Linux. Because we use the IBOS components in Linux, we believe that this performance difference occurs from overhead in the IBOS kernel. To test this hypothesis, we ran a number of micro benchmarks on the two systems and we believe that the overhead is due to contention for spinlocks in the L4 IPC implementation. The net effect of this contention is that heavy use of network processes requires heavy use of IPC, which adds latency to all IPC messages and slows down the overall system. However, the IBOS-Linux results for these experiments show that this slow down is not fundamental and can be fixed with a more mature kernel implementation.

Overall, the page load latency experiments show that even with a prototype implementation of IBOS, our architecture will not slow down the browsing speed significantly for the web sites we tested.

7 Additional related work

7.1 Alternative kernel architectures

Operating systems designed to reduce the trusted computing base for applications are not new. For example, several recent OSes propose using information flow to allow applications to specify information flow policies that are enforced by a thin kernel [18, 57, 33]; KeyKOS [12], EROS [45], and seL4 [32] provide capability support using a small kernel; and Microkernels [24, 27, 28] push typical OS components into user space. In IBOS, we apply these principles to a new application – the web browser – and include support for user interface components and window manager operations. Also, these previous approaches support general purpose security mechanisms, like information flow and capabilities, and shared resources and device drivers are part of the TCB. The IBOS security policy is specific to web browsers, and although this is less general, we can track this policy to hardware abstractions and can remove drivers and other shared components from our TCB.

Both Exokernels [19, 31] and L4 [27] rethink low-layer software abstractions. In both projects, they advocate exposing abstractions that are close to the underlying hardware to enable applications to customize for improved performance. In IBOS we build on these previous works – in fact we use the L4Ka::Pistachio L4 [8] MMU abstractions and message passing implementation directly. However, the key difference between our work and L4 and Exokernel is that we expose high-level application abstractions at our lowest layer of software, not low-level hardware abstractions. Our focus is on making web browsers more secure and the system software we use to accomplish this improved security.

7.2 Browser security

A number of recent papers have proposed new browser architectures including SubOS [29, 30], safe web programs [44], OP [26], Chrome [11, 43], Gazelle [52], and ServiceOS [38]. Although the browser portion of IBOS does resemble some of these works, they all run on top of commodity OSes and include complex libraries and window managers in their TCB, something that IBOS avoids by focusing on the OS architecture of our system.

The webOS from Palm [40] and the upcoming ChromeOS from Google [25] run a web browser on top of a Linux kernel. ChromeOS includes kernel hardening using trusted boot, mandatory access controls, and sandboxing mechanisms for reducing the attack surface

of their system. However, ChromeOS and IBOS have fundamentally different design philosophies. ChromeOS starts with a large and complex system and tries to remove and restrict the unused and unneeded portions of the system. In contrast, IBOS starts with a clean slate and only adds to our system functionality needed for our browser. Although our approach does require implementing from scratch low-level software and fitting device drivers to a new driver model, the end result has 2 to 3 orders of magnitude fewer lines of code in the TCB, while still retaining nearly all of the same functionality.

In the Tahoma browser [15], the authors propose using virtual machine monitors (VMMs) to enable web applications to specify code that runs on the client. Tahoma uses server-side manifests to specify the security policy for the downloaded code and the VMM enforces this security policy. Tahoma does expose a few browser abstractions from their VMM to help manage UI elements and network connections, but operates mostly on hardware-level abstractions. Because Tahoma operates on hardware-level abstractions, Tahoma is unable to provide full backwards-compatible web semantics from the VMM and more fine-grained protection for browsers, such as isolating `iframes` embedded in a web application. Also, many modern VMMs use a full-blown commodity OS in a privileged virtual machine or host OS for driver support, leaving tens of millions of lines of code in the TCB potentially.

7.3 Device driver security

Device driver security has focused on three main topics. First, several projects focus on restricting driver access to I/O ports and device access to main memory via DMA. For example, RVM uses a software-only approach to restrict DMA access of devices [55], SVA prevents the OS from accessing driver registers via memory mapped I/O through memory safety checks [16], and Mungi [35] relies on using a hardware IOMMU to limit which memory regions are accessible from devices. Second, system designers isolate drivers from the rest of the system. This isolation can be achieved by running drivers in user-mode, which has been a staple of Microkernel systems [24, 36, 28], using software to protect the OS from kernel drivers [20, 58], or by using page table protections within the OS [49, 48]. The driver security architecture in IBOS differs from these approaches because our system provides fine-grained protection for individual requests within a shared driver in addition to isolating the driver from the rest of the system.

7.4 Secure window managers

A number of recent projects have looked at reducing the TCB for window managers. For example DoPE [21] and Nitpicker [22] move widget rendering from the server to the client, leaving the server to only manage shared buffers. CMW [56], EWS [46], and TrustGraph [39] also use clients for rendering, but are able to apply capabilities and mandatory access control policies to application user-interface elements. In IBOS, we deprecate the general window notion of modern computer systems in favor of the simpler browser chrome and tab motif, allowing us to track our security policies down to the underlying graphics hardware on our system.

8 Conclusions

In this paper, we presented IBOS, an operating system and web browser co-designed to reduce drastically the trusted computing base for web browsers and to simplify browsing systems. To achieve this improvement, we built IBOS with browser abstractions as first-class OS abstractions and removed traditional shared system components and services from its TCB. With our new architecture, we showed that IBOS enforced traditional and novel security policies, and we argued that the overall system security and usability could withstand successful attacks on device drivers, browser components, or traditional applications. Our experimental results showed that IBOS added little overhead when compared to today's high-performance browsers running on fast and mature commodity operating systems.

Acknowledgment

We would like to thank Brad Chen, Steve Gribble, and Hank Levy for their feedback on our security analysis. We would also like to thank our shepherd Nickolai Zeldovich, Anthony Cozzie, and Matt Hicks who provided valuable feedback on this paper. This research was funded in part by NSF grants CNS 0834738 and CNS 0831212, grant N0014-09-1-0743 from the Office of Naval Research, AFOSR MURI grant FA9550-09-01-0539, and by a grant from the Internet Services Research Center (ISRC) of Microsoft Research.

References

[1] CVE - Common Vulnerabilities and Exposures (CVE). `http://cve.mitre.org`.

[2] Gecko plugin API reference. `https://developer.mozilla.org/en/Gecko_Plugin_API_Reference`.

[3] Qt - A Cross-platform application and UI. `http://qt.nokia.com/`.

[4] Symantec internet security threat report april 2010. http://www.symantec.com/business/theme.jsp?themeid=threatreport.

[5] The WebKit Open Source Project. http://webkit.org/.

[6] uClibc. http://www.uclibc.org/.

[7] Qt labs blogs: So long and thanks for the blit, 2008. http://labs.trolltech.com/blogs/2008/10/22/so-long-and-thanks-for-the-blit/.

[8] L4Ka::Pistachio microkernel, 2010. http://l4ka.org/projects/pistachio.

[9] ANDERSON, J. P. Computer security technology planning study. Tech. rep., HQ Electronic Systems Division (AFSC), October 1972. ESD-TR-73-51.

[10] BARTH, A., CABALLERO, J., AND SONG, D. Secure content sniffing for web browsers or how to stop papers from reviewing themselves. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2009).

[11] BARTH, A., JACKSON, C., REIS, C., AND THE GOOGLE CHROME TEAM. The security architecture of the chromium browser, 2008. http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf.

[12] BOMBERGER, A. C., FRANTZ, W. S., HARDY, A. C., HARDY, N., LANDAU, C. R., AND SHAPIRO, J. S. The KeyKOS nanokernel architecture. In *Proceedings of the Workshop on Microkernels and Other Kernel Architectures* (Berkeley, CA, USA, 1992), USENIX Association, pp. 95–112.

[13] CHEN, S., MESEGUER, J., SASSE, R., WANG, H. J., AND WANG, Y.-M. A systematic approach to uncover security flaws in GUI logic. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (May 2007), pp. 71–85.

[14] CHEN, S., ROSS, D., AND WANG, Y.-M. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)* (2007), pp. 2–11.

[15] COX, R. S., HANSEN, J. G., GRIBBLE, S. D., AND LEVY, H. M. A safety-oriented platform for web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (May 2006), pp. 350–364.

[16] CRISWELL, J., GEOFFRAY, N., AND ADVE, V. Memory safety for low-level software/hardware interactions. In *Proceedings of the Eighteenth Usenix Security Symposium* (August 2009).

[17] DUNKELS, A., WOESTENBERG, L., MANSLEY, K., AND MONOSES, J. lwIP embedded TCP/IP stack. http://savannah.nongnu.org/projects/lwip/, 2004.

[18] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. Labels and event processes in the asbestos operating system. In *SOSP '05: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2005), ACM, pp. 17–30.

[19] ENGLER, D. R., KAASHOEK, M. F., AND JR., J. O. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 1995 Symposium on Operating Systems Principles* (December 1995), pp. 251–266.

[20] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. Xfi: software guards for system address spaces. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 75–88.

[21] FESKE, N., AND HÄRTIG, H. DOpE - a window server for real-time and embedded systems. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium* (Washington, DC, USA, 2003), IEEE Computer Society, p. 74.

[22] FESKE, N., AND HELMUTH, C. A Nitpicker's guide to a minimal-complexity secure GUI. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 85–94.

[23] GARFINKEL, T. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the 2003 Network and Distributed System Security Symposium (NDSS)* (February 2003).

[24] GOLUB, D., DEAN, R., FORIN, A., AND RASHID, R. Unix as an Application Program. In *Proceedings of the 1990 USENIX Summer Conference* (1990).

[25] GOOGLE INC. Chromium OS, 2010. http://www.chromium.org/chromium-os.

[26] GRIER, C., TANG, S., AND KING, S. T. Secure web browsing with the OP web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (May 2008), pp. 402–416.

[27] HÄRTIG, H., HOHMUTH, M., LIEDTKE, J., WOLTER, J., AND SCHÖNBERG, S. The performance of μ-kernel-based systems. In *SOSP '97: Proceedings of the sixteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1997), ACM, pp. 66–77.

[28] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. MINIX 3: a highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev. 40*, 3 (2006), 80–89.

[29] IOANNIDIS, S., AND BELLOVIN, S. M. Building a secure web browser. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (June 2001).

[30] IOANNIDIS, S., BELLOVIN, S. M., AND SMITH, J. Sub-operating systems: A new approach to application security. In *SIGOPS European Workshop* (September 2002).

[31] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., BRICEÑO, H. M., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on exokernel systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1997), ACM, pp. 52–65.

[32] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: formal verification of an os kernel. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 207–220.

[33] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard OS abstractions. In *SOSP '07: Proceedings of twenty-first ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2007), ACM, pp. 321–334.

[34] LAWRENCE, E. Combating clickjacking with x-frame-options, March 2010. http://blogs.msdn.com/b/ieinternals/archive/2010/03/30/combating-clickjacking-with-x-frame-options.aspx.

[35] LESLIE, B., AND HEISER, G. Towards untrusted device drivers. Tech. rep., UNSW-CSE-TR-0303, 2003.

[36] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GOTZ, S. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 2004 Symposium*

on Operating Systems Design and Implementation (OSDI) (December 2004).

[37] MOSHCHUK, A., BRAGIN, T., GRIBBLE, S. D., AND LEVY, H. M. A crawler-based study of spyware on the web. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)* (February 2006).

[38] MOSHCHUK, A., AND WANG, H. J. Resource Management for Web Applications in ServiceOS. Tech. rep., Microsoft Research, May 2010.

[39] OKHRAVI, H., AND NICOL, D. M. Trustgraph: Trusted graphics subsystem for high assurance systems. In *ACSAC '09: Proceedings of the 2009 Annual Computer Security Applications Conference* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 254–265.

[40] PALM INC. webOS, 2010. `http://opensource.palm.com`.

[41] PROVOS, N., MAVROMMATIS, P., RAJAB, M. A., AND MONROSE, F. All your iFRAMEs point to us. In *Proceedings of the 17th Usenix Security Symposium* (July 2008), pp. 1–15.

[42] PROVOS, N., MCNAMEE, D., MAVROMMATIS, P., WANG, K., AND MODADUGU, N. The ghost in the browser: Analysis of Web-based malware. In *Proceedings of the 2007 Workshop on Hot Topics in Understanding Botnets (HotBots)* (April 2007).

[43] REIS, C., AND GRIBBLE, S. D. Isolating web programs in modern browser architectures. In *Proceedings of the 2009 EuroSys conference* (2009).

[44] REIS, C., GRIBBLE, S. D., AND LEVY, H. M. Architectural principles for safe web programs. In *Proceedings of the Sixth Workshop on Hot Topics in Networks (HotNets)* (November 2007).

[45] SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. EROS: a fast capability system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles* (New York, NY, USA, 1999), ACM, pp. 170–185.

[46] SHAPIRO, J. S., VANDERBURGH, J., NORTHUP, E., AND CHIZMADIA, D. Design of the EROS trusted window system. In *Proceedings of the 13th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association, pp. 12–12.

[47] SINGH, K., MOSHCHUK, A., WANG, H. J., AND LEE, W. On the incoherencies in web browser access control policies. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2010).

[48] SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering Device Drivers. In *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI)* (December 2004).

[49] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM, pp. 207–222.

[50] SYMANTEC INC. Symantec global Internet security threat report: Trends for 2008, April 2009. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`.

[51] TAN, L., ZHANG, X., MA, X., XIONG, W., AND ZHOU, Y. AutoISES: Automatically inferring security specifications and detecting violations. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security '08)* (July-August 2008).

[52] WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 2009 USENIX Security Symposium* (August 2009).

[53] WANG, Y.-M., BECK, D., JIANG, X., ROUSSEV, R., VERBOWSKI, C., CHEN, S., AND KING, S. Automated Web Patrol with Strider HoneyMonkeys: Finding Web sites that exploit browser vulnerabilities. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)* (February 2006).

[54] WHEELER, D. SLOCcount, 2009. `http://www.dwheeler.com/sloccount/`.

[55] WILLIAMS, D., REYNOLDS, P., WALSH, K., SIRER, E. G., AND SCHNEIDER, F. B. Device driver safety through a reference validation mechanism. In *OSDI 08: Proceedings of the 8th symposium on operating systems design and implementation* (2008).

[56] WOODWARD, J. P. Security requirementes for systems high and compartemented mode workstations. Tech. rep., MITRE Corp., 1987. MTR 9992.

[57] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 263–278.

[58] ZHOU, F., CONDIT, J., ANDERSON, Z., BAGRAK, I., ENNALS, R., HARREN, M., NECULA, G., AND BREWER, E. Safedrive: safe and recoverable extensions using language-based techniques. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 45–60.

FlexSC: Flexible System Call Scheduling with Exception-Less System Calls

Livio Soares
University of Toronto

Michael Stumm
University of Toronto

Abstract

For the past 30+ years, system calls have been the *de facto* interface used by applications to request services from the operating system kernel. System calls have almost universally been implemented as a *synchronous* mechanism, where a special processor instruction is used to yield user-space execution to the kernel. In the first part of this paper, we evaluate the performance impact of traditional synchronous system calls on system intensive workloads. We show that synchronous system calls negatively affect performance in a significant way, primarily because of pipeline flushing and pollution of key processor structures (e.g., TLB, data and instruction caches, etc.).

We propose a new mechanism for applications to request services from the operating system kernel: *exception-less system calls*. They improve processor efficiency by enabling flexibility in the scheduling of operating system work, which in turn can lead to significantly increased temporal and spacial locality of execution in both user and kernel space, thus reducing pollution effects on processor structures. Exception-less system calls are particularly effective on multicore processors. They primarily target highly threaded server applications, such as Web servers and database servers.

We present FlexSC, an implementation of exception-less system calls in the Linux kernel, and an accompanying user-mode thread package (FlexSC-Threads), binary compatible with POSIX threads, that translates legacy synchronous system calls into exception-less ones *transparently* to applications. We show how FlexSC improves performance of Apache by up to 116%, MySQL by up to 40%, and BIND by up to 105% while requiring no modifications to the applications.

1 Introduction

System calls are the *de facto* interface to the operating system kernel. They are used to request services offered by, and implemented in the operating system kernel. While

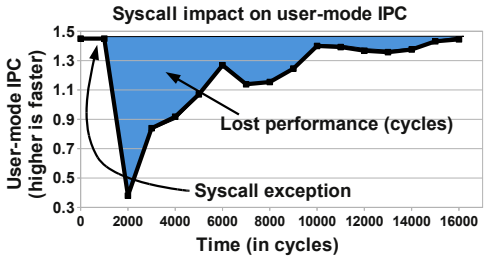


Figure 1: User-mode instructions per cycles (IPC) of Xalan (from SPEC CPU 2006) in response to a system call exception event, as measured on an Intel Core i7 processor.

different operating systems offer a variety of different services, the basic underlying system call mechanism has been common on all commercial multiprocessed operating systems for decades. System call invocation typically involves writing arguments to appropriate registers and then issuing a special machine instruction that raises a synchronous exception, immediately yielding user-mode execution to a kernel-mode exception handler. Two important properties of the traditional system call design are that: (1) a processor exception is used to communicate with the kernel, and (2) a synchronous execution model is enforced, as the application expects the completion of the system call before resuming user-mode execution. Both of these effects result in performance inefficiencies on modern processors.

The increasing number of available transistors on a chip (Moore's Law) has, over the years, led to increasingly sophisticated processor structures, such as superscalar and out-of-order execution units, multi-level caches, and branch predictors. These processor structures have, in turn, led to a large increase in the performance *potential* of software, but at the same time there is a widening gap between the performance of efficient software and the performance of inefficient software, primarily due to the increasing disparity of accessing different processor resources (e.g., registers vs. caches vs. memory). Server and system-intensive workloads, which are of particular

interest in our work, are known to perform well below the potential processor throughput [11, 12, 19]. Most studies attribute this inefficiency to the lack of locality. *We claim that part of this lack of locality, and resulting performance degradation, stems from the current synchronous system call interface.*

Synchronous implementation of system calls negatively impacts the performance of system intensive workloads, both in terms of the *direct* costs of mode switching and, more interestingly, in terms of the *indirect* pollution of important processor structures which affects both user-mode and kernel-mode performance. A motivating example that quantifies the impact of system call pollution on application performance can be seen in Figure 1. It depicts the user-mode instructions per cycles (kernel cycles and instructions are ignored) of one of the SPEC CPU 2006 benchmarks (Xalan) immediately before and after a `pwrite` system call. There is a significant drop in instructions per cycle (IPC) due to the system call, and it takes up to 14,000 cycles of execution before the IPC of this application returns to its previous level. As we will show, this performance degradation is mainly due to interference caused by the kernel on key processor structures.

To improve locality in the execution of system intensive workloads, we propose a new operating system mechanism: the **exception-less system call**. An exception-less system call is a mechanism for requesting kernel services that does not require the use of synchronous processor exceptions. In our implementation, system calls are issued by writing kernel requests to a reserved **syscall page**, using normal memory store operations. The actual execution of system calls is performed asynchronously by special in-kernel **syscall threads**, which post the results of system calls to the syscall page after their completion.

Decoupling the system call execution from its invocation creates the possibility for flexible system call scheduling, offering optimizations along two dimensions. The first optimization allows for the deferred batch execution of system calls resulting in increased temporal locality of execution. The second provides the ability to execute system calls on a separate core, in parallel to executing user-mode threads, resulting in spatial, per core locality. In both cases, system call threads become a simple, but powerful abstraction.

One interesting feature of the proposed decoupled system call model is the possibility of *dynamic* core specialization in multicore systems. Cores can become temporarily specialized for either user-mode or kernel-mode execution, depending on the current system load. We describe how the operating system kernel can dynamically adapt core specialization to the demands of the workload.

One important challenge of our proposed system is how to best use the exception-less system call interface. One option is to rewrite applications to directly interface with the exception-less system call mechanism. We believe the lessons learned by the systems community with event-driven servers indicate that directly using exception-less system calls would be a daunting software engineering task. For this reason, we propose a new M-on-N threading package (M user-mode threads executing on N kernel-visible threads, with $M >> N$). The main purpose of this threading package is to harvest independent system calls by switching threads, in user-mode, whenever a thread invokes a system call.

This research makes the following contributions:

- We quantify, at fine granularity, the impact of synchronous mode switches and system call execution on the micro-architectural processor structures, as well as on the overall performance of user-mode execution.
- We propose a new operating system mechanism, the exception-less system call, and describe an implementation, FlexSC[1], in the Linux kernel.
- We present a M-on-N threading system, compatible with PThreads, that transparently uses the new exception-less system call facility.
- We show how exception-less system calls coupled with our M-on-N threading system improves performance of important system-intensive highly threaded workloads: Apache by up to 116%, MySQL by to 40%, and BIND by up to 105%.

2 The (Real) Costs of System Calls

In this section, we analyze the performance costs associated with a traditional, synchronous system call. We analyze these costs in terms of mode switch time, the system call footprint, and the effect on user-mode and kernel-mode IPC. We used the Linux operating system kernel and an Intel Nehalem (Core i7) processor, along with its performance counters to obtain our measurements. However, we believe the lessons learned are applicable to most modern high-performance processors[2] and other operating system kernels.

2.1 Mode Switch Cost

Traditionally, the performance cost attributed to system calls is the *mode switch time*. The mode switch time consists of the time necessary to execute the appropriate system call instruction in user-mode, resuming execution in an elevated protection domain (kernel-mode), and the return of control back to user-mode. Modern processors implement the mode switch as a processor exception: flushing the user-mode pipeline, saving a few registers onto the

[1] Pronounced as "flex" (/'fleks/).

[2] Experiments performed on an older PowerPC 970 processor yielded similar insights than the ones presented here.

Syscall	Instructions	Cycles	IPC	i-cache	d-cache	L2	L3	d-TLB
`stat`	4972	13585	0.37	32	186	660	2559	21
`pread`	3739	12300	0.30	32	294	679	2160	20
`pwrite`	5689	31285	0.18	50	373	985	3160	44
`open+close`	6631	19162	0.34	47	240	900	3534	28
`mmap+munmap`	8977	19079	0.47	41	233	869	3913	7
`open+write+close`	9921	32815	0.30	78	481	1462	5105	49

Table 1: System call footprint of different processor structures. For the processors structures (caches and TLB), the numbers represent number of entries evicted; the cache line for the processor is of 64-bytes. i-cache and d-cache refer to the instruction and data sections of the L1 cache, respectively. The d-TLB represents the data portion of the TLB.

kernel stack, changing the protection domain, and redirecting execution to the registered exception handler. Subsequently, return from exception is necessary to resume execution in user-mode.

We measured the mode switch time by implementing a new system call, `gettsc` that obtains the time stamp counter of the processor and immediately returns to user-mode. We created a simple benchmark that invoked `gettsc` 1 billion times, recording the time-stamp before and after each call. The difference between each of the three time-stamps identifies the number of cycles necessary to enter and leave the operating system kernel, namely 79 cycles and 71 cycles, respectively. The total round-trip time for the `gettsc` system call is modest at 150 cycles, being less than the latency of a memory access that misses the processor caches (250 cycles on our machine).[3]

2.2 System Call Footprint

The mode switch time, however, is only part of the cost of a system call. During kernel-mode execution, processor structures including the L1 data and instruction caches, translation look-aside buffers (TLB), branch prediction tables, prefetch buffers, as well as larger unified caches (L2 and L3), are populated with kernel specific state. The replacement of user-mode processor state by kernel-mode processor state is referred to as the processor state *pollution* caused by a system call.

To quantify the pollution caused by system calls, we used the Core i7 hardware performance counters (HPC). We ran a high instruction per cycle (IPC) workload, Xalan, from the SPEC CPU 2006 benchmark suite that is known to invoke few system calls. We configured an HPC to trigger infrequently (once every 10 million user-mode instructions) so that the processor structures would be dominated with application state. We then set up the HPC exception handler to execute specific system calls, while measuring the replacement of application state in the processor structures caused by kernel execution (but not by the performance counter exception handler itself).

[3]For all experiments presented in this paper, user-mode applications execute in 64-bit mode and when using synchronous system calls, use the "syscall" x86_64 instruction, which is currently the default in Linux.

Table 1 shows the footprint on several processor structures for three different system calls and three system call combinations. The data shows that, even though the number of i-cache lines replaced is modest (between 2 and 5 KB), the number of d-cache lines replaced is significant. Given that the size of the d-cache on this processor is 32 KB, we see that the system calls listed pollute at least half of the d-cache, and almost all of the d-cache in the "open+write+close" case. The 64 entry first level d-TLB is also significantly polluted by most system calls. Finally, it is interesting to note that the system call impact on the L2 and L3 caches is larger than on the L1 caches, primarily because the L2 and L3 caches use more aggressive prefetching.

2.3 System Call Impact on User IPC

Ultimately, the most important measure of the real cost of system calls is the performance impact on the application. To quantify this, we executed an experiment similar to the one described in the previous subsection. However, instead of measuring kernel-mode events, we only measured user-mode instructions per cycle (IPC), ignoring all kernel execution. Ideally, user-mode IPC should not decrease as a result of invoking system calls, since the cycles and instructions executed as part of the system call are ignored in our measurements. In practice, however, user-mode IPC is affected by two sources of overhead:

Direct: The processor exception associated with the system call instruction that flushes the processor pipeline.

Indirect: System call pollution on the processor structures, as quantified in Table 1.

Figures 2 and 3 show the degradation in user-mode IPC when running Xalan (from SPEC CPU 2006) and SPECJBB, respectively, given different frequencies of `pwrite` calls. These benchmarks were chosen since they have been created to avoid significant use of system services, and should spend only 1-2% of time executing in kernel-mode. The graphs show that different workloads can have different sensitivities to system call pollution. Xalan has a baseline user-mode IPC of 1.46, but the IPC degrades by up to 65% when executing a `pwrite` every 1,000-2,000 instructions, yielding an IPC between 0.58 and 0.50.

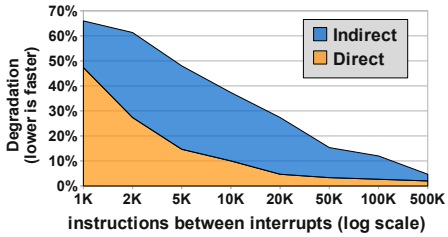


Figure 2: System call (`pwrite`) impact on user-mode IPC as a function of system call frequency for Xalan.

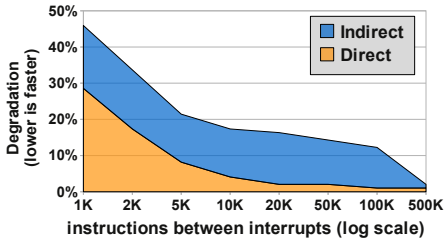


Figure 3: System call (`pwrite`) impact on user-mode IPC as a function of system call frequency for SPEC JBB.

SPEC-JBB has a slightly lower baseline of 0.97, but still observes a 45% degradation of user-mode IPC.

The figures also depict the breakdown of user-mode IPC degradation due to direct and indirect costs. The degradation due to the direct cost was measured by issuing a null system call, while the indirect portion is calculated subtracting the direct cost from the degradation measured when issuing a `pwrite` system call. For high frequency system call invocation (once every 2,000 instructions, or less), the direct cost of raising an exception and subsequent flushing of the processor pipeline is the largest source of user-mode IPC degradation. However, for medium frequencies of system call invocation (once per 2,000 to 100,000 instructions), the *indirect* cost of system calls is the dominant source of user-mode IPC degradation.

To understand the implication of these results on typical server workloads, it is necessary to quantify the system call frequency of these workloads. The average user-mode instruction count between consecutive system calls for three popular server workloads are shown in Table 2. For this frequency range in Figures 2 and 3 we observe user-mode IPC performance degradation between 20% and 60%. While the excecution of the server workloads listed in Table 2 is not identical to that of Xalan or SPEC-JBB, the data presented here indicates that server workloads suffer from significant performance degradation due to processor pollution of system calls.

Workload (server)	Instructions per Syscall
DNSbench (BIND)	2445
ApacheBench (Apache)	3368
Sysbench (MySQL)	12435

Table 2: The average number of instructions executed on different workloads before issuing a syscall.

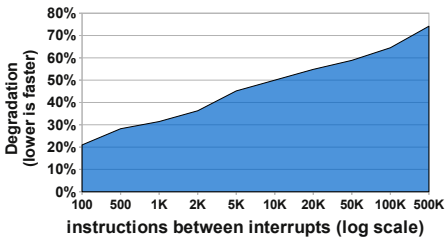


Figure 4: System call (`pwrite`), impact on kernel-mode IPCs for x as a function of system call frequency.

2.4 Mode Switching Cost on Kernel IPC

The lack of locality due to frequent mode switches also negatively affects kernel-mode IPC. Figure 4 shows the impact of different system call frequencies on the kernel-mode IPC. As expected, the performance trend is opposite to that of user-mode execution. The more frequent the system calls, the more kernel state is maintained in the processor.

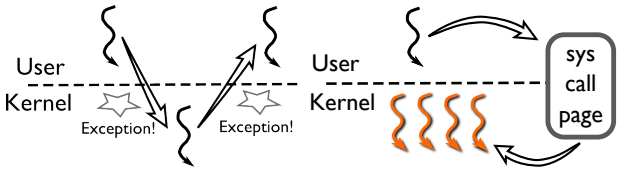
Note that the kernel-mode IPC listed in Table 1 for different system calls ranges from 0.18 to 0.47, with an average of 0.32. This is significantly lower than the 1.47 and 0.97 user-mode IPC for Xalan and SPEC-JBB, respectively; up to *8x* slower.

3 Exception-Less System Calls

To address (and partially eliminate) the performance impact of traditional, synchronous system calls on system intensive workloads, we propose a new operating system mechanism called **exception-less system call**. Exception-less system call is a mechanism for requesting kernel services that does not require the use of synchronous processor exceptions. The key benefit of exception-less system calls is the flexibility in scheduling system call execution, ultimately providing improved locality of execution of both user and kernel code. We explore two use cases:

System call batching: Delaying the execution of a series of system calls and executing them in batches minimizes the frequency of switching between user and kernel execution, eliminating some of the mode switch overhead and allowing for improved *temporal locality*. This improves both the direct and indirect costs of system calls.

Core specialization: In multicore systems, exception-less system calls allow a system call to be scheduled on a core different than the one on which the system call was invoked. Scheduling system calls on a separate processor core allows for improved *spatial locality* and with it lower



(a) Traditional, sync. system call (b) Exception-less system call

Figure 5: Illustration of synchronous and exception-less system call invocation. The left diagram shows the sequential nature of exception-based system calls, while the right diagram depicts exception-less user and kernel communication through shared memory.

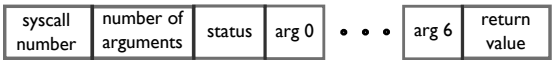


Figure 6: 64-byte syscall entry from the syscall page.

indirect costs. In an ideal scenario, no mode switches are necessary, eliminating the direct cost of system calls.

The design of exception-less system calls consists of two components: (1) an exception-less interface for user-space threads to register system calls, along with (2) an in-kernel threading system that allows the delayed (asynchronous) execution of system calls, without interrupting or blocking the thread in user-space.

3.1 Exception-Less Syscall Interface

The interface for exception-less system calls is simply a set of memory pages that is shared amongst user and kernel space. The shared memory page, henceforth referred to as **syscall page**, is organized to contain exception-less system call entries. Each entry contains space for the request status, system call number, arguments, and return value.

With traditional synchronous system calls, invocation occurs by populating predefined registers with system call information and issuing a specific machine instruction that immediately raises an exception. In contrast, to issue an exception-less system call, the user-space threads must find a free entry in the syscall page and populate the entry with the appropriate values using regular store instructions. The user-space thread can then continue executing without interruption. It is the responsibility of the user-space thread to later verify the completion of the system call by reading the status information in the entry. None of these operations, issuing a system call or verifying its completion, causes exceptions to be raised.

3.2 Syscall Pages

Syscall pages can be viewed as a table of syscall entries, each containing information specific to a single system call request, including the system call number, arguments, status (free/submitted/busy/done), and the result (Figure 6). In our 64-bit implementation, we have organized each entry to occupy 64 bytes. This size comes from the Linux ABI which allows any system call to have up to 6 arguments, and a return value, totalling 56 bytes. Although the remaining 3 fields (syscall number, status and number of arguments) could be packed in less than the remaining 8 bytes, we selected 64 bytes because 64 is a divisor of popular cache line sizes of today's processor.

To issue an exception-less system call, the user-space thread must find an entry in one of its syscall pages that contain a *free* status field. It then writes the syscall number and arguments to the entry. Lastly, the status field is changed to *submitted*[4], indicating to the kernel that the request is ready for execution. The thread must then check the status of the entry until it becomes *done*, consume the return value, and finally set the status of the entry to *free*.

3.3 Decoupling Execution from Invocation

Along with the exception-less interface, the operating system kernel must support delayed execution of system calls. Unlike exception-based system calls, the exception-less system call interface does not result in an explicit kernel notification, nor does it provide an execution stack. To support decoupled system call execution, we use a special type of kernel thread, which we call **syscall thread**. Syscall threads always execute in kernel mode, and their sole purpose is to pull requests from syscall pages and execute them on behalf of the user-space thread. Figure 5 illustrates the difference between traditional synchronous system calls, and our proposed split system call model.

The combination of the exception-less system call interface and independent syscall threads allows for great flexibility in the scheduling the execution of system calls. Syscall threads may wake up only after user-space is unable to make further progress, in order to achieve temporal locality of execution on the processor. Orthogonally, syscall threads can be scheduled on a different processor core than that of the user-space thread, allowing for spatial locality of execution. On modern multicore processors, cache to cache communication is relatively fast (in the order of 10s of cycles), so communicating the entries of syscall pages from a user-space core to a kernel core, or vice-versa, should only cause a small number of processor stalls.

3.4 Implementation – FlexSC

Our implementation of the exception-less system call mechanism is called **FlexSC** (Flexible System Call) and was prototyped as an extension to the Linux kernel. Although our implementation was influenced by a mono-

[4]User-space must update the status field last, with an appropriate memory barrier, to prevent the kernel from selecting incomplete syscall entries to execute.

lithic kernel architecture, we believe that most of our design could be effective with other kernel architectures, e.g., exception-less micro-kernel IPCs, and hypercalls in a paravirtualized environment.

We have implemented FlexSC for the x86_64 and PowerPC64 processor architectures. Porting FlexSC to other architectures is trivial; a single function is needed, which moves arguments from the syscall page to appropriate registers, according to the ABI of the processor architecture. Two new system calls were added to Linux as part of FlexSC, `flexsc_register` and `flexsc_wait`.

`flexsc_register()` This system call is used by processes that wish to use the FlexSC facility. Making this registration procedure explicit is not strictly necessary, as processes can be registered with FlexSC upon creation. We chose to make it explicit mainly for convenience of prototyping, giving us more control and flexibility in user-space. One legitimate reason for making registration explicit is to avoid the extra initialization overheads incurred for processes that do not use exception-less system calls.

Invocation of the `flexsc_register` system call must use the traditional, exception-based system call interface to avoid complex bootstrapping; however, since this system call needs to execute only once, it does not impact application performance. Registration involves two steps: mapping one or more syscall pages into user-space virtual memory space, and spawning one syscall thread per entry in the syscall pages.

`flexsc_wait()` The decoupled execution model of exception-less system calls creates a challenge in user-space execution, namely what to do when the user-space thread has nothing more to execute and is waiting on pending system calls. With the proposed execution model, the OS kernel loses the ability to determine when a user-space thread should be put to sleep. With synchronous system calls, this is simply achieved by putting the thread to sleep while it is executing a system call if the call blocks waiting for a resource.

The solution we adopted is to require that the user explicitly communicate to the kernel that it cannot progress until one of the issued system calls completes by invoking the `flexsc_wait` system call. We implemented `flexsc_wait` as an exception-based system call, since execution should be synchronously directed to the kernel. FlexSC will later wake up the user-space thread when at least one of posted system calls are complete.

3.5 Syscall Threads

Syscall threads is the mechanism used by FlexSC to allow for exception-less execution of system calls. The Linux system call execution model has influenced some implementation aspects of syscall threads in FlexSC: (1) the virtual address space in which system call execution occurs is the address space of the corresponding process, and (2) the current thread context can be used to block execution should a necessary resource not be available (for example, waiting for I/O).

To resolve the virtual address space requirement, syscall threads are created during `flexsc_register`. Syscall threads are thus "cloned" from the registering process, resulting in threads that share the original virtual address space. This allows the transfer of data from/to user-space with no modification to Linux's code.

FlexSC would ideally never allow a syscall thread to sleep. If a resource is not currently available, notification of the resource becoming available should be arranged, and execution of the next pending system call should begin. However, implementing this behavior in Linux would require significant changes and a departure from the basic Linux architecture. Instead, we adopted a strategy that allows FlexSC to maintain the Linux thread blocking architecture, as well as requiring only minor modifications (3 lines of code) to Linux context switching code, by creating multiple syscall threads for each process that registers with FlexSC.

In fact, FlexSC spawns as many syscall threads as there are entries available in the syscall pages mapped in the process. This provisions for the worst case where every pending system call blocks during execution. Spawning hundreds of syscall threads may seem expensive, but Linux in-kernel threads are typically much lighter weight than user threads: all that is needed is a `task_struct` and a small, 2-page, stack for execution. All the other structures (page table, file table, etc.) are shared with the user process. In total, only 10KB of memory is needed per syscall thread.

Despite spawning multiple threads, only *one* syscall thread is active per application and core at any given point in time. If system calls do not block all the work is executed by a single syscall thread, while the remaining ones sleep on a work-queue. When a syscall thread needs to block, for whatever reason, immediately before it is put to sleep, FlexSC notifies the work-queue. Another thread wakes-up and immediately starts executing the next system call. Later, when resources become free, current Linux code wakes up the waiting thread (in our case, a syscall thread), and resumes its execution, so it can post its result to the syscall page and return to wait in the FlexSC work-queue.

3.6 FlexSC Syscall Thread Scheduler

FlexSC implements a syscall thread scheduler that is responsible for determining when and on which core system calls will execute. This scheduler is critical to performance, as it influences the locality of user and kernel execution.

On a single-core environment, the FlexSC scheduler assumes the user-space will attempt to post as many exception-less system calls as possible, and subsequently call `flexsc_wait()`. The FlexSC scheduler then wakes up an available syscall thread that starts executing the first system call. If the system call does not block, the same syscall thread continues to execute the next submitted syscall entry. If the execution of a syscall thread blocks, the currently scheduled syscall thread notifies the scheduler to wake another thread to continue to execute more system calls. The scheduler does not wake up the user-space thread until all available system calls have been issued, and have either finished or are currently blocked with at least one system call having been completed. This is done to minimize the number of mode switches to user-space.

For multicore execution, the scheduler biases execution of syscall threads on a subset of available cores, dynamically specializing cores according to the workload requirements. In our current implementation, this is done by attempting to schedule syscall threads using a predetermined, static list of cores. Upon a scheduling decision, the first core on the list is selected. If a syscall thread of a process is currently running on that core, the next core on the list is selected as the target. If the selected core is not currently executing a syscall thread, an inter-processor interrupt is sent to the remote core, signalling that it must wake a syscall thread.

As previously described, there is never more than one syscall thread concurrently executing per core, for a given process. However in the multicore case, for the same process, there can be as many syscall threads as cores concurrently executing on the entire system. To avoid cache-line contention of syscall pages amongst cores, before a syscall thread begins executing calls from a syscall page, it *locks* the page until all its submitted calls have been issued. Since FlexSC processes typically map multiple syscall pages, each core on the system can schedule a syscall thread to work independently, executing calls from different syscall pages.

4 System Calls Galore – FlexSC-Threads

Exception-less system calls present a significant change to the semantics of the system call interface with potentially drastic implications for application code and programmers. Programming using exception-less system calls directly is more complex than using synchronous system calls, as they do not provide the same, easy-to-reason-about sequentiality. In fact, our experience is that programming using exception-less system calls is akin to event-driven programming, which has itself been criticized for being a complex programming model [21]. The main difference is that with exception-less system calls, not only are I/O related calls scheduled for future completion, *any* system calls can be requested, verified for completion, and handled, as if it were an asynchronous event.

To address the programming complexities, we propose the use of exception-less system calls in two different modes that might be used depending on the concurrency model adopted by the programmer. We argue that if used according to our recommendations, exception-less system calls should pose *no more* complexity than their synchronous counter-parts.

4.1 Event-driven Servers, a Case for Hybrid Execution

For event-driven systems, we advocate a *hybrid* approach where both synchronous and exception-less system calls coexist. System calls that are executed in performance critical paths of applications should use exception-less calls while all other calls should be synchronous. After all, there is no good justification to make a simple *getpid()* complex to program.

Event-driven servers already have their code structured so that performance critical paths of execution are split into three parts: request event, wait for completion and handle event. Adapting an event-driven server to use exception-less system calls, for the already considered events, should be straightforward. However, we have not yet attempted to evaluate the use of exception-less system calls in an event-driven program, and leave this as future work.

4.2 FlexSC-Threads

Multiprocessing has become the default for computation on servers. With the emergence and ubiquity of multicore processors, along with projection of future chip manufacturing technologies, it is unlikely that this trend will reverse in the medium future. For this reason, and because of its relative simplicity vis-a-vis event-based programming, we believe that the multithreading concurrency model will continue to be the norm.

In this section, we describe the design and implementation of **FlexSC-Threads**, a threading package that transforms legacy synchronous system calls into exception-less ones *transparently* to applications. It is intended for server-type applications with many user-mode threads, such as Apache or MySQL. FlexSC-Threads is compliant with POSIX Threads, and binary compatible with NPTL [8], the default Linux thread library. As a result, Linux multi-threaded programs work with FlexSC-Threads "out of the box" without modification or recompilation.

FlexSC-Threads uses a simple M-on-N threading model (M user-mode threads executing on N kernel-

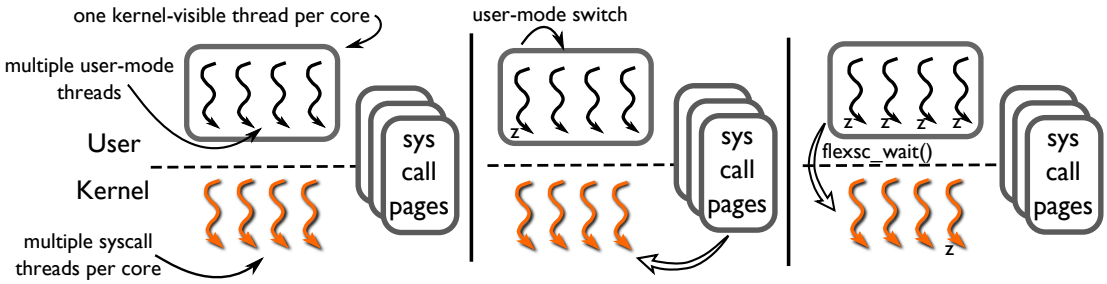


Figure 7: The left-most diagram depicts the components of FlexSC-Threads pertaining to a single core. Each core executes a pinned kernel-visible thread, which in turn can multiplex multiple user-mode threads. Multiple syscall pages, and consequently syscall threads, are also allocated (and pinned) per core. The middle diagram depicts a user-mode thread being preempted as a result of issuing a system call. The right-most diagram depicts the scenario where all user-mode threads are waiting for system call requests; in this case FlexSC-Threads library synchronously invokes `flexsc_wait()` to the kernel.

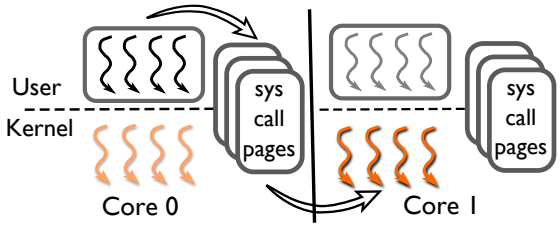


Figure 8: Multicore example. Opaque threads are active, while grayed-out threads are inactive. Syscall pages are accessible to both cores, as we run using shared-memory, leveraging the fast on-chip communication of multicores.

visible threads). We rely on the ability to perform user-mode thread switching solely in user-space to transparently transform legacy synchronous calls into exception-less ones. This is done as follows:

1. We redirect to our library each *libc* call that issues a legacy system call. Typically, applications do not directly embed code to issue system calls, but instead call wrappers in the dynamically loaded *libc*. We use the dynamic loading capabilities of Linux to redirect execution of such calls to our library.
2. FlexSC-Threads then post the corresponding exception-less system call to a syscall page and switch to another user-mode thread that is ready.
3. If we run out of ready user-mode threads, FlexSC checks the syscall page for any syscall entries that have been completed, waking up the appropriate user-mode thread so it can obtain the result of the completed system call.
4. As a last resort, `flexsc_wait()` is called, putting the kernel visible thread to sleep until one of the pending system calls has completed.

FlexSC-Threads implements multicore support by creating a *single* kernel visible thread *per* core available to the process, and pinning each kernel visible thread to a specific core. Multiple user-mode threads multiplex execution on the kernel visible thread. Since kernel-visitble threads only block when there is no more available work, there is no need to create more than one kernel visible thread per core. Figure 7 depicts the components of FlexSC-Threads and how they interact during execution.

As an optimization, we have designed FlexSC-Threads to register a private set of syscall pages *per* kernel visible thread (i.e., per core). Since syscall pages are private to each core, there is no need to synchronize their access with costly atomic instructions. The FlexSC-Threads user-mode scheduler implements a simple form of cooperative scheduling, with system calls acting as yield points. Consequently, syscall pages behave as lock-free single-producer (kernel-visible thread) and single-consumer (syscall thread) data structures.

From the kernel side, although syscall threads are pinned to specific cores, they do not only execute system call requests from syscall pages registered to that core. An example of this is shown in Figure 8, where user-mode threads execute on core 0, while syscall threads running on core 1 are satisfying system call requests.

It is important to note that FlexSC-Threads relies on a large number of independent user-mode threads to post concurrent exception-less system calls. Since threads are executing independently, there is no constraint on ordering or serialization of system call execution (thread-safety constraints should be enforced at the application level and is orthogonal to the system call execution model). FlexSC-Threads leverages the independent requests to efficiently schedule operating system work on single or multicore systems. For this reason, highly threaded workloads, such as internet/network servers, are ideal candidates for FlexSC-Threads.

5 Experimental Evaluation

We first present the results of a microbenchmark that shows the overhead of the basic exception-less system

Component	Specification
Cores	4
Cache line	64 B for all caches
Private L1 i-cache	32 KB, 3 cycle latency
Private L1 d-cache	32 KB, 4 cycle latency
Private L2 cache	512 KB, 11 cycle latency
Shared L3 cache	8 MB, 35-40 cycle latency
Memory	250 cycle latency (avg.)
TLB (L1)	64 (data) + 64 (instr.) entries
TLB (L2)	512 entries

Table 3: Characteristics of the 2.3GHz Core i7 processor.

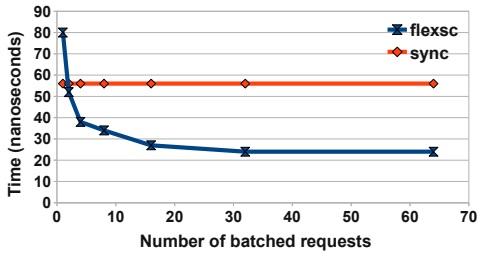


Figure 9: Exception-less system call cost on a single-core.

call mechanism, and then we show the performance of two popular server applications, Apache and MySQL, transparently using exception-less system calls through FlexSC-Threads. Finally, we analyze the sensitivity of the performance of FlexSC to the number of system call pages.

FlexSC was implemented in the Linux kernel, version 2.6.33. The baseline line measurements we present were collected using unmodified Linux (same version), and the default native POSIX threading library (NPTL). We identify the baseline configuration as "**sync**", and the system with exception-less system calls as "**flexsc**".

The experiments presented in this section were run on an Intel Nehalem (Core i7) processor with the characteristics shown in Table 3. The processor has 4 cores, each with 2 hyper-threads. We disabled the hyper-threads, as well as the "TurboBoost" feature, for all our experiments to more easily analyze the measurements obtained.

For the Apache and MySQL experiments, requests were generated by a remote client connected to our test machine through a 1 Gbps network, using a dedicated router. The client machine contained a dual core Core2 processor, running the same Linux installation as the test machine, and was not CPU or network constrained in any of the experiments.

All values reported in our evaluation represent the average of 5 separate runs.

5.1 Overhead

The overhead of executing an exception-less system call involves switching to a syscall thread, de-marshalling arguments from the appropriate syscall page entry, switching back to the user-thread, and retrieving the return value from the syscall page entry. To measure this overhead, we created a micro-benchmark that successively invokes a `getppid()` system call. Since the user and kernel footprints of this call is small, the time measured corresponds to the *direct* cost of issuing system calls.

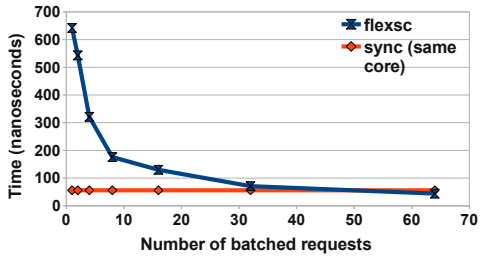


Figure 10: Exception-less system call cost, in the worst case, for remote core execution.

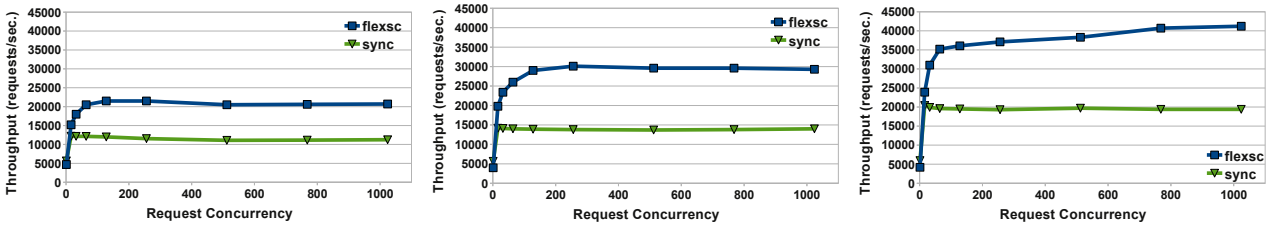
We varied the number of batched system calls, in the exception-less case, to verify if the direct costs are amortized when batching an increasing number of calls. The results obtained executing on a single core are shown in Figure 9. The baseline time, show as a horizontal line, is the time to execute an exception-based system call on a single core. Executing a single exception-less system call on a single core is 43% slower than a synchronous call. However, when batching 2 or more calls there is no overhead, and when batching 32 or more calls, the execution of each call is up to 130% faster than a synchronous call.

We also measured the time to execute system calls on a remote core (Figure 10). In addition to the single core operations, remote core execution entails sending an inter-processor interrupt (IPI) to wake up the remote syscall thread. In the remote core case, the time to issue a single exception-less system call can be more than 10 times slower than a synchronous system call on the same core. This measurement represents a worst case scenario when there is no currently executing syscall thread. Despite the high overhead, the overhead on remote core execution is recouped when batching 32 or more system calls.

5.2 Apache

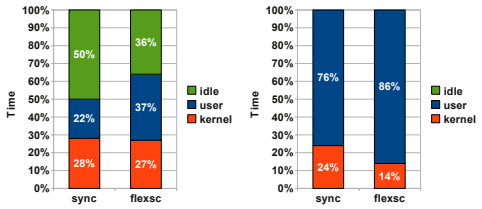
We used Apache version 2.2.15 to evaluate the performance of FlexSC-Threads. Since FlexSC-Threads is binary compatible with NPTL, we used the same Apache binary for both FlexSC and Linux/NPTL experiments. We configured Apache to use a different maximum number of spawned threads for each case. The performance of Apache running on NPTL degrades with too many threads, and we experimentally determined that 200 was optimal for our workload and hence used that configuration for the NPTL case. For the FlexSC-Threads case, we raised the maximum number of threads to 1000.

The workload we used was ApacheBench, a HTTP



(a) 1 Core (b) 2 Cores (c) 4 Cores

Figure 11: Comparison of Apache throughput of Linux/NPTL and FlexSC executing on 1, 2 and 4 cores.



(a) Apache (b) MySQL

Figure 12: Breakdown of execution time of Apache and MySQL workloads on 4 cores.

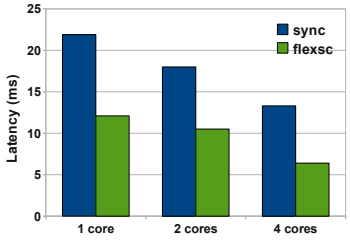


Figure 13: Comparison of Apache latency of Linux/NPTL and FlexSC executing on 1, 2 and 4 cores, with 256 concurrent requests.

workload generator that is distributed with Apache. It is designed to stress-test the Web server determining the number of requests per second that can be serviced, with varying number of concurrent requests.

Figure 11 shows the results of Apache running on 1, 2 and 4 cores. For the single core experiments, FlexSC employs system call batching, and for the multicore experiments it additionally dynamically redirects system calls to maximize core locality. The results show that, except for a very low number of concurrent requests, FlexSC outperforms Linux/NPTL by a wide margin. With system call batching alone (1 core case), we observe a throughput improvement of up to 86%. The 2 and 4 core experiments show that FlexSC achieves up to 116% throughput improvement, showing the added benefit of dynamic core specialization.

Table 4 shows the effects of FlexSC on the microarchitectural state of the processor while running Apache. It displays various processor metrics, collected using hardware performance counters during execution with 512 concurrent requests. The most important metric listed is the instruction per cycles (IPC) of the user and kernel mode for the different setups, as it summarizes the efficiency of execution. The other values listed are normalized values using *misses per kilo-instructions* (MPKI). MPKI is a widely used normalization method that makes it easy to compare values obtained from different executions.

The most efficient execution of the four listed in the table is FlexSC on 1 core, yielding an IPC of 0.94 on both kernel and user execution, which is 95–108% higher than for NPTL. While the FlexSC execution of Apache on 4 cores is not as efficient as the single core case, with an average IPC of 0.75, there is still an 71% improvement, on average, over NPTL.

Most metrics we collected are significantly improved with FlexSC. Of particular importance are the performance critical structures that have a high MPKI value on NPTL such as d-cache, i-cache, and L2 cache. The better use of these microarchitectural structures effectively demonstrates the premise of this work, namely that exception-less system calls can improve processor efficiency. The only structure which observes more misses on FlexSC is the user-mode TLB. We are currently investigating the reason for this.

There is an interesting disparity between the throughput improvement (94%) and the IPC improvement (71%) in the 4 core case. The difference comes from the added benefit of localizing kernel execution with core specialization. Figure 12a shows the time breakdown of Apache executing on 4 cores. FlexSC execution yields significantly less idle time than the NPTL execution.[5] The reduced idle time is a consequence of lowering the contention on a specific kernel semaphore. Linux protects address spaces with a per address-space read-write semaphore (`mmap_sem`). Profiling shows that every Apache thread allocates and frees memory for serving requests, and both of these operations require the semaphore to be held with write permission. Further, the network code in Linux invokes `copy_user()`, which transfers data in and out of the user address-space. This function verifies that the user-space memory is indeed valid, and to do so acquires

[5]The execution of Apache on 1 or 2 core did not present idle time.

Apache	User							Kernel						
Setup	**IPC**	L3	L2	d-cache	i-cache	TLB	Branch	**IPC**	L3	L2	d-cache	i-cache	TLB	Branch
sync (1 core)	**0.48**	3.7	68.9	63.8	130.8	7.7	20.9	**0.45**	1.4	80.0	78.2	159.6	4.6	15.7
flexsc (1 core)	**0.94**	1.7	27.5	35.3	41.3	8.8	12.6	**0.94**	1.0	15.8	31.6	45.2	3.3	11.2
sync (4 cores)	**0.45**	3.9	64.6	67.9	127.6	9.6	20.2	**0.43**	4.4	49.5	73.8	124.9	4.4	15.2
flexsc (4 cores)	**0.74**	1.0	37.5	55.5	49.4	19.3	13.0	**0.76**	1.5	19.1	50.2	63.7	4.2	11.6

Table 4: Micro-architectural breakdown of Apache execution on uni- and quad-core setups. All values shown, except for IPC, are normalized using misses per kilo-instruction (MPKI): therefore, lower numbers yield more efficient execution and higher IPC.

the semaphore with read permissions. In the NPTL case, threads from all 4 cores compete on this semaphore, resulting in 50% idle time. With FlexSC, kernel code is dynamically scheduled to run predominantly on 2 out of the 4 cores, halving the contention to this resource, eliminating 38% of the original idle time.

Another important metric for servicing Web requests besides throughput is latency of individual requests. One might intuitively expect that latency of requests to be higher under FlexSC because of batching and asynchronous servicing of system calls, but the opposite is the case. Figure 13 shows the average latency of requests when processing 256 concurrent requests (other concurrency levels showed similar trends). The results show that Web requests on FlexSC are serviced within 50-60% of the time needed on NPTL, on average.

5.3 MySQL

In the previous section, we demonstrated the effectiveness of FlexSC running on a workload with a significant proportion of kernel time. In this section, we experiment with OLTP on MySQL, a workload for which the proportion of kernel execution is smaller (roughly 25%). Our evaluation used MySQL version 5.5.4 with an InnoDB backend engine, and as in the Apache evaluation, we used the same binary for running on NPTL and on FlexSC. We also used the same configuration parameters for both the NPTL and FlexSC experiments, after tuning them for the best NPTL performance.

To generate requests to MySQL, we used the *sysbench* system benchmark utility. Sysbench was created for benchmarking MySQL processor performance and contains an OLTP inspired workload generator. The benchmark allows executing concurrent requests by spawning multiple client threads, connecting to the server, and sequentially issuing SQL queries. To handle the concurrent clients, MySQL spawns a user-level thread per connection. At the end, sysbench reports the number of transactions per second executed by the database, as well as average latency information. For these experiments, we used a database with 5M rows, resulting in 1.2 GB of data. Since we were interested in stressing the CPU component of MySQL, we disabled synchronous transactions to disk. Given that the configured database was small enough to fit in memory, the workload presented no idle time due to disk I/O.

Figure 14 shows the throughput numbers obtained on 1, 2 and 4 cores when varying the number of concurrent client threads issuing requests to the MySQL server.[6] For this workload, system batching on one core provides modest improvements: up to 14% with 256 concurrent requests. On 2 and 4 cores, however, we see that FlexSC provides a consistent improvement with 16 or more concurrent clients, achieving up to 37%-40% higher throughput.

Table 5 contains the microarchitectural processor metrics collected for the execution of MySQL. Because MySQL invokes the kernel less frequently than Apache, kernel execution yields high miss rates, resulting in a low IPC of 0.33 on NPTL. In the single core case, FlexSC does not greatly alter the execution of user-space, but increases kernel IPC by 36%. FlexSC allows the kernel to reuse state in the processor structures, yielding lower misses across most metrics. In the case of 4 cores, FlexSC also improves the performance of user-space IPC by as much as 30%, compared to NPTL. Despite making less of an impact in the kernel IPC than in single core execution, there is still a 25% kernel IPC improvement over NPTL.

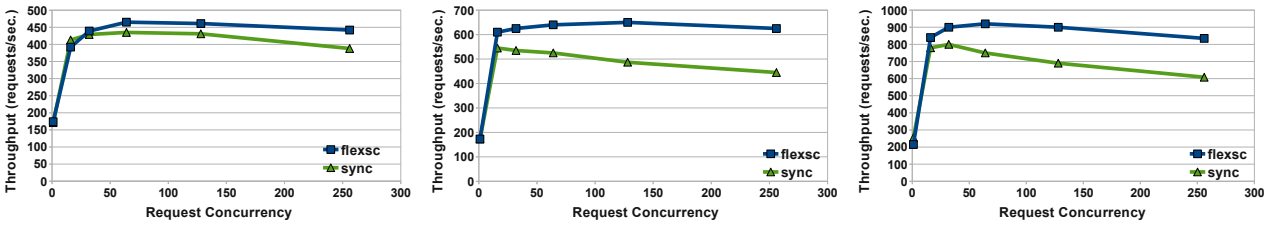
Figure 15 shows the average latencies of individual requests for MySQL execution with 256 concurrent clients. As is the case with Apache, the latency of requests on FlexSC is improved over execution on NPTL. Requests on FlexSC are satisfied within 70-88% of the time used by requests on NPTL.

5.4 Sensitivity Analysis

In all experiments presented so far, FlexSC was configured to have 8 system call pages per core, allowing up to 512 concurrent exception-less system calls per core.

Figure 16 shows the sensitivity of FlexSC to the number of available syscall entries. It depicts the throughput of Apache, on 1 and 4 cores, while servicing 2048 concurrent requests per core, so that there would always be more requests available than syscall entries. Uni-core performance approaches its best with 200 to 250 syscall entries

[6] For both NPTL and FlexSC, increasing the load on MySQL yields peak throughput between 32 and 128 concurrent clients after which throughput degrades. The main reason for performance degradation is the costly and coarse synchronization used in MySQL. MySQL and Linux kernel developers have observed similar performance degradation.



(a) 1 Core (b) 2 Cores (c) 4 Cores

Figure 14: Comparison of MySQL throughput of Linux/NPTL and FlexSC executing on 1, 2 and 4 cores.

MySQL	User							Kernel						
Setup	**IPC**	L3	L2	d-cache	i-cache	TLB	Branch	**IPC**	L3	L2	d-cache	i-cache	TLB	Branch
sync (1 core)	**1.12**	0.6	21.1	34.8	24.2	3.8	7.8	**0.33**	16.5	125.2	209.6	184.9	3.9	17.4
flexsc (1 core)	**1.10**	0.8	19.6	36.3	23.6	5.4	6.9	**0.45**	23.2	55.1	131.9	86.5	3.7	13.6
sync (4 cores)	**0.55**	3.7	15.8	25.2	18.9	3.1	5.9	**0.36**	16.6	78.0	147.0	120.0	3.6	15.7
flexsc (4 cores)	**0.72**	2.7	16.7	30.6	20.9	4.1	6.5	**0.45**	18.4	46.6	104.4	63.5	2.5	11.5

Table 5: Micro-architectural breakdown of MySQL execution on uni- and quad-core setups. All values shown, except for IPC, are normalized using misses per kilo-instruction (MPKI): therefore, lower numbers yield more efficient execution and higher IPC.

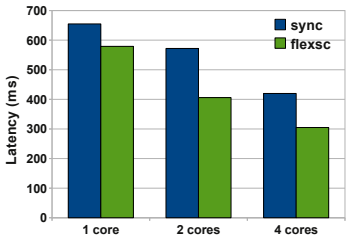


Figure 15: Comparison of MySQL latency of Linux/NPTL and FlexSC executing on 1, 2 and 4 cores, with 256 concurrent requests.

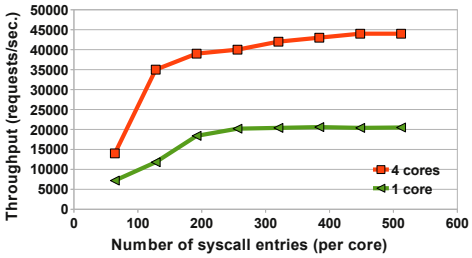


Figure 16: Execution of Apache on FlexSC-Threads, showing the performance sensitivity of FlexSC to different number of syscall pages. Each syscall page contains 64 syscall entries.

(3 to 4 syscall pages), while quad-core execution starts to plateau with 300 to 400 syscall entries (6 to 7 syscall pages).

It is particularly interesting to compare Figure 16 with figures 9 and 10. The direct cost of mode switching, exemplified by the micro-benchmark, has a lesser effect on performance when compared to the indirect cost of mixing user- and kernel-mode execution.

6 Related Work

6.1 System Call Batching

The idea of batching calls in order to save crossings has been extensively explored in the systems community. Specific to operating systems, *multi-calls* are used in both operating systems and paravirtualized hypervisors as a mechanism to address the high overhead of mode switching. Cassyopia is a compiler targeted at rewriting programs to collect many independent system calls, and submitting them as a single multi-call [18]. An interesting technique in Cassyopia, which could be eventually explored in conjunction with FlexSC, is the concept of a *looped multi-call* where the result of one system call can be automatically fed as an argument to another system call in the same multi-call. In the context of hypervisors, both Xen and VMware currently support a special multi-call hypercall feature [4][20].

An important difference between multi-calls and exception-less system calls is the level of flexibility exposed. The multi-call proposals do not investigate the possibility of parallel execution of system calls, or address the issue of blocking system calls. In multi-calls, system calls are executed sequentially; each system call must complete before a subsequent can be issued. With exception-less system calls, system calls can be executed in parallel, and in the presence of blocking, the next call can execute immediately.

6.2 Locality of Execution and Multicores

Several researchers have studied the effects of operating system execution on application performance [1, 3, 7, 6, 11, 13]. Larus and Parkes also identified processor inef-

ficiencies of server workloads, although not focusing on the interaction with the operating system. They proposed Cohort Scheduling to efficiently execute staged computations to improve locality of execution [11].

Techniques such as Soft Timers [3] and Lazy Receiver Processing [9] also address the issue of locality of execution, from the other side of the compute stack: handling device interrupts. Both techniques describe how to limit processor interference associated with interrupt handling, while not impacting the latency of servicing requests.

Most similar to the multicore execution of FlexSC is Computation Spreading proposed by Chakraborty et. al [6]. They introduced processor modifications to allow for hardware migration of threads, and evaluated the effects on migrating threads upon entering the kernel to specialize cores. Their simulation-based results show an improvement of up to 20% on Apache, however, they explicitly do not model TLBs and provide for fast thread migration between cores. On current hardware, synchronous thread migration between cores requires a costly inter-processor interrupt.

Recently, both Corey and Factored Operating System (fos) have proposed dedicating cores for specific operating system functionality [24, 25]. There are two main differences between the core specialization possible with these proposals and FlexSC. First, both Corey and fos require a micro-kernel design of the operating system kernel in order to execute specific kernel functionality on dedicated cores. Second, FlexSC can dynamically adapt the proportion of cores used by the kernel, or cores shared by user and kernel execution, depending on the current workload behavior.

Explicit off-loading of select OS functionality to cores has also been studied for performance [15, 16] and power reduction in the presence of single-ISA heterogeneous multicores [14]. While these proposals rely on expensive inter-processor interrupts to offload system calls, we hope FlexSC can provide for a more efficient, and flexible, mechanism that can be used by such proposals.

6.3 Non-blocking Execution

Past research on improving system call performance has focused extensively on blocking versus non-blocking behavior. Typically researchers have analyzed the use of threading, event-based (non-blocking), and hybrid systems for achieving high performance on server applications [2, 10, 17, 21, 22, 23]. Capriccio described techniques to improve performance of user-level thread libraries for server applications [22]. Specifically, Behren et al. showed how to efficiently manage thread stacks, minimizing wasted space, and propose resource aware scheduling to improver server performance. For an extensive performance comparison of thread-based and event-driven Web server architectures we refer the reader to [17].

Finally, the Linux community has proposed a generic mechanism for implementing non-blocking system calls, which is call asynchronous system calls [5]. In their proposal, system calls are still exception-based, and tentatively execute synchronously. Like scheduler activations, if a blocking condition is detected, they utilize a "syslet" thread to block, allowing the user thread to continue execution.

The main difference between many of the proposals for non-blocking execution and FlexSC is that none of the non-blocking system call proposals completely decouple the invocation of the system call from its execution. As we have discussed, the flexibility resulting from this decoupling is crucial for efficiently exploring optimizations such as system call batching and core specialization.

7 Concluding Remarks

In this paper, we introduced the concept of exception-less system calls that decouples system call invocation from execution. This allows for flexible scheduling of system call execution which in turn enables system call batching and dynamic core specialization that both improve locality in a significant way. System calls are issued by writing kernel requests to a reserved syscall page using normal store operations, and they are executed by special in-kernel syscall threads, which then post the results to the syscall page.

In fact, the concept of exception-less system calls originated as a mechanism for low-latency communication between user and kernel-space with hyper-threaded processors in mind. We had hoped that communicating directly through the shared L1 cache would be much more efficient than mode switching. However, the measurements presented in Section 2 made it clear that mixing user and kernel-mode execution on the same core would not be efficient for server class workloads. In future work we intend to study how to exploit exception-less system calls as a communication mechanism in hyper-threaded processors.

We presented our implementation of FlexSC, a Linux kernel extension, and FlexSC-Threads, a M-on-N threading package that is binary compatible with NPTL and that transparently transforms synchronous system calls into exception-less ones. With this implementation, we demonstrated how FlexSC improves throughput of Apache by up to 116% and MySQL by up to 40% while requiring no modifications to the applications. We believe these two workloads are representative of other highly threaded server workloads that would benefit from FlexSC. For example, experiments with the BIND DNS server demonstrated throughput improvements of between 30% and 105% depending on the concurrency of requests.

In the current implementation of FlexSC, syscall threads process system call requests in no specific order, opportunistically issuing calls as they are posted on syscall pages. The asynchronous execution model, however, would allow for different selection algorithms. For example, syscall threads could sort the requests to consecutively execute requests of the same type, potentially yielding greater locality of execution. Also, system calls that perform I/O could be prioritized so as to issue them as early as possible. Finally, if a large number of cores are available, cores could be dedicated to specific system call types to promote further locality gains.

8 Acknowledgements

This work was supported in part by Discovery Grant funding from the Natural Sciences and Engineering Research Council (NSERC) of Canada. We would like to thank the feedback from the OSDI reviewers, and to Emmett Witchel for shepherding our paper. Special thanks to Ioana Burcea for encouraging the work in its early stages, and the Computer Systems Lab members (University of Toronto), as well as Benjamin Gamsa, for insightful comments on the work and drafts of this paper.

References

[1] AGARWAL, A., HENNESSY, J., AND HOROWITZ, M. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst. 6*, 4 (1988), 393–431.

[2] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Trans. Comput. Syst. 10*, 1 (1992), 53–79.

[3] ARON, M., AND DRUSCHEL, P. Soft timers: efficient microsecond software timer support for network processing. *ACM Trans. Comput. Syst. (TOCS) 18*, 3 (2000), 197–228.

[4] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 164–177.

[5] BROWN, Z. Asynchronous system calls. In *Proceedings of the Ottawa Linux Symposium (OLS)* (2007), pp. 81–85.

[6] CHAKRABORTY, K., WELLS, P. M., AND SOHI, G. S. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006), pp. 283–292.

[7] CHEN, J. B., AND BERSHAD, B. N. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)* (1993), pp. 120–133.

[8] DREPPER, U., AND MOLNAR, I. The Native POSIX Thread Library for Linux. Tech. rep., RedHat Inc, 2003. http://people.redhat.com/drepper/nptl-design.pdf.

[9] DRUSCHEL, P., AND BANGA, G. Lazy receiver processing (LRP): a network subsystem architecture for server systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (1996), pp. 261–275.

[10] ELMELEEGY, K., CHANDA, A., COX, A. L., AND ZWAENEPOEL, W. Lazy asynchronous I/O for event-driven servers. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)* (2004), pp. 21–21.

[11] LARUS, J., AND PARKES, M. Using Cohort-Scheduling to Enhance Server Performance. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)* (2002), pp. 103–114.

[12] LI, T., JOHN, L. K., SIVASUBRAMANIAM, A., VIJAYKRISHNAN, N., AND RUBIO, J. Understanding and Improving Operating System Effects in Control Flow Prediction. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2002), pp. 68–80.

[13] MOGUL, J. C., AND BORG, A. The Effect of Context Switches on Cache Performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1991), pp. 75–84.

[14] MOGUL, J. C., MUDIGONDA, J., BINKERT, N., RANGANATHAN, P., AND TALWAR, V. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro 28*, 3 (2008), 26–41.

[15] NELLANS, D., BALASUBRAMONIAN, R., AND BRUNVAND, E. OS execution on multi-cores: is out-sourcing worthwhile? *SIGOPS Oper. Syst. Rev. 43*, 2 (2009), 104–105.

[16] NELLANS, D., SUDAN, K., BRUNVAND, E., AND BALASUBRAMONIAN, R. Improving Server Performance on Multi-Cores via Selective Off-loading of OS Functionality. In *Sixth Annual Workshorp on the Interaction between Operating Systems and Computer Architecture (WIOSCA)* (2010), pp. 13–20.

[17] PARIAG, D., BRECHT, T., HARJI, A., BUHR, P., SHUKLA, A., AND CHERITON, D. R. Comparing the performance of Web server architectures. In *Proceedings of the 2nd European Conference on Computer Systems (Eurosys)* (2007), pp. 231–243.

[18] RAJAGOPALAN, M., DEBRAY, S. K., HILTUNEN, M. A., AND SCHLICHTING, R. D. Cassyopia: compiler assisted system optimization. In *Proceedings of the 9th conference on Hot Topics in Operating Systems (HotOS)* (2003), pp. 18–18.

[19] REDSTONE, J. A., EGGERS, S. J., AND LEVY, H. M. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2000), pp. 245–256.

[20] VMWARE. *VMWare Virtual Machine Interface Specification*. http://www.vmware.com/pdf/vmi_specs.pdf.

[21] VON BEHREN, R., CONDIT, J., AND BREWER, E. Why Events Are A Bad Idea (for high-concurrency servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems (HotOS)* (2003).

[22] VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 268–281.

[23] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (2001), pp. 230–243.

[24] WENTZLAFF, D., AND AGARWAL, A. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev. 43*, 2 (2009), 76–85.

[25] WICKIZER, S. B., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).

Finding a needle in Haystack: Facebook's photo storage

Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, Peter Vajgel,
Facebook Inc.
{doug, skumar, hcli, jsobel, pv}@facebook.com

Abstract: This paper describes Haystack, an object storage system optimized for Facebook's Photos application. Facebook currently stores over 260 billion images, which translates to over 20 petabytes of data. Users upload one billion new photos (~60 terabytes) each week and Facebook serves over one million images per second at peak. Haystack provides a less expensive and higher performing solution than our previous approach, which leveraged network attached storage appliances over NFS. Our key observation is that this traditional design incurs an excessive number of disk operations because of metadata lookups. We carefully reduce this per photo metadata so that Haystack storage machines can perform all metadata lookups in main memory. This choice conserves disk operations for reading actual data and thus increases overall throughput.

1 Introduction

Sharing photos is one of Facebook's most popular features. To date, users have uploaded over 65 billion photos making Facebook the biggest photo sharing website in the world. For each uploaded photo, Facebook generates and stores four images of different sizes, which translates to over 260 billion images and more than 20 petabytes of data. Users upload one billion new photos (~60 terabytes) each week and Facebook serves over one million images per second at peak. As we expect these numbers to increase in the future, photo storage poses a significant challenge for Facebook's infrastructure.

This paper presents the design and implementation of Haystack, Facebook's photo storage system that has been in production for the past 24 months. Haystack is an object store [7, 10, 12, 13, 25, 26] that we designed for sharing photos on Facebook where data is written once, read often, never modified, and rarely deleted. We engineered our own storage system for photos because traditional filesystems perform poorly under our workload.

In our experience, we find that the disadvantages of a traditional POSIX [21] based filesystem are directories and per file metadata. For the Photos application most of this metadata, such as permissions, is unused and thereby wastes storage capacity. Yet the more significant cost is that the file's metadata must be read from disk into memory in order to find the file itself. While insignificant on a small scale, multiplied over billions of photos and petabytes of data, accessing metadata is the throughput bottleneck. We found this to be our key problem in using a network attached storage (NAS) appliance mounted over NFS. Several disk operations were necessary to read a single photo: one (or typically more) to translate the filename to an inode number, another to read the inode from disk, and a final one to read the file itself. In short, using disk IOs for metadata was the limiting factor for our read throughput. Observe that in practice this problem introduces an additional cost as we have to rely on content delivery networks (CDNs), such as Akamai [2], to serve the majority of read traffic.

Given the disadvantages of a traditional approach, we designed Haystack to achieve four main goals:

High throughput and low latency. Our photo storage systems have to keep up with the requests users make. Requests that exceed our processing capacity are either ignored, which is unacceptable for user experience, or handled by a CDN, which is expensive and reaches a point of diminishing returns. Moreover, photos should be served quickly to facilitate a good user experience. Haystack achieves high throughput and low latency by requiring at most one disk operation per read. We accomplish this by keeping all metadata in main memory, which we make practical by dramatically reducing the per photo metadata necessary to find a photo on disk.

Fault-tolerant. In large scale systems, failures happen every day. Our users rely on their photos being available and should not experience errors despite the inevitable server crashes and hard drive failures. It may happen that an entire datacenter loses power or a cross-country link is severed. Haystack replicates each photo in geographically distinct locations. If we lose a machine we introduce another one to take its place, copying data for redundancy as necessary.

Cost-effective. Haystack performs better and is less

expensive than our previous NFS-based approach. We quantify our savings along two dimensions: Haystack's cost per terabyte of usable storage and Haystack's read rate normalized for each terabyte of usable storage[1]. In Haystack, each usable terabyte costs ∼**28%** less and processes ∼**4x** more reads per second than an equivalent terabyte on a NAS appliance.

Simple. In a production environment we cannot overstate the strength of a design that is straight-forward to implement and to maintain. As Haystack is a new system, lacking years of production-level testing, we paid particular attention to keeping it simple. That simplicity let us build and deploy a working system in a few months instead of a few years.

This work describes our experience with Haystack from conception to implementation of a production quality system serving billions of images a day. Our three main contributions are:

- Haystack, an object storage system optimized for the efficient storage and retrieval of billions of photos.
- Lessons learned in building and scaling an inexpensive, reliable, and available photo storage system.
- A characterization of the requests made to Facebook's photo sharing application.

We organize the remainder of this paper as follows. Section 2 provides background and highlights the challenges in our previous architecture. We describe Haystack's design and implementation in Section 3. Section 4 characterizes our photo read and write workload and demonstrates that Haystack meets our design goals. We draw comparisons to related work in Section 5 and conclude this paper in Section 6.

2 Background & Previous Design

In this section, we describe the architecture that existed before Haystack and highlight the major lessons we learned. Because of space constraints our discussion of this previous design elides several details of a production-level deployment.

2.1 Background

We begin with a brief overview of the typical design for how web servers, content delivery networks (CDNs), and storage systems interact to serve photos on a popular site. Figure 1 depicts the steps from the moment when a user visits a page containing an image until she downloads that image from its location on disk. When visiting a page the user's browser first sends an HTTP request to a web server which is responsible for generating the markup for the browser to render. For each image the web server constructs a URL directing the browser to a location from which to download the data. For popular sites this URL often points to a CDN. If the CDN has the image cached then the CDN responds immediately with the data. Otherwise, the CDN examines the URL, which has enough information embedded to retrieve the photo from the site's storage systems. The CDN then updates its cached data and sends the image to the user's browser.

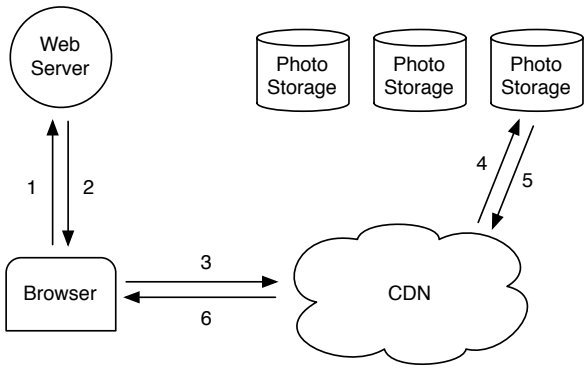


Figure 1: Typical Design

2.2 NFS-based Design

In our first design we implemented the photo storage system using an NFS-based approach. While the rest of this subsection provides more detail on that design, the major lesson we learned is that CDNs by themselves do not offer a practical solution to serving photos on a social networking site. CDNs do effectively serve the hottest photos— profile pictures and photos that have been recently uploaded—but a social networking site like Facebook also generates a large number of requests for less popular (often older) content, which we refer to as the *long tail*. Requests from the long tail account for a significant amount of our traffic, almost all of which accesses the backing photo storage hosts as these requests typically miss in the CDN. While it would be very convenient to cache all of the photos for this long tail, doing so would not be cost effective because of the very large cache sizes required.

Our NFS-based design stores each photo in its own file on a set of commercial NAS appliances. A set of

[1]The term 'usable' takes into account capacity consumed by factors such as RAID level, replication, and the underlying filesystem

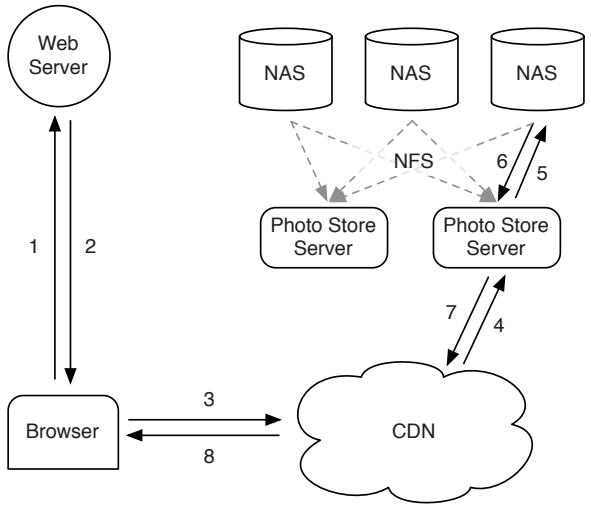


Figure 2: NFS-based Design

machines, Photo Store servers, then mount all the volumes exported by these NAS appliances over NFS. Figure 2 illustrates this architecture and shows Photo Store servers processing HTTP requests for images. From an image's URL a Photo Store server extracts the volume and full path to the file, reads the data over NFS, and returns the result to the CDN.

We initially stored thousands of files in each directory of an NFS volume which led to an excessive number of disk operations to read even a single image. Because of how the NAS appliances manage directory metadata, placing thousands of files in a directory was extremely inefficient as the directory's blockmap was too large to be cached effectively by the appliance. Consequently it was common to incur more than 10 disk operations to retrieve a single image. After reducing directory sizes to hundreds of images per directory, the resulting system would still generally incur 3 disk operations to fetch an image: one to read the directory metadata into memory, a second to load the inode into memory, and a third to read the file contents.

To further reduce disk operations we let the Photo Store servers explicitly cache file handles returned by the NAS appliances. When reading a file for the first time a Photo Store server opens a file normally but also caches the filename to file handle mapping in memcache [18]. When requesting a file whose file handle is cached, a Photo Store server opens the file directly using a custom system call, `open_by_filehandle`, that we added to the kernel. Regrettably, this file handle cache provides only a minor improvement as less popular photos are less likely to be cached to begin with. One could argue that an approach in which all file handles are stored in memcache might be a workable solution. However, that only addresses part of the problem as it relies on the NAS appliance having all of its inodes in main memory, an expensive requirement for traditional filesystems. The major lesson we learned from the NAS approach is that focusing only on caching—whether the NAS appliance's cache or an external cache like memcache—has limited impact for reducing disk operations. The storage system ends up processing the long tail of requests for less popular photos, which are not available in the CDN and are thus likely to miss in our caches.

2.3 Discussion

It would be difficult for us to offer precise guidelines for when or when not to build a custom storage system. However, we believe it still helpful for the community to gain insight into why we decided to build Haystack.

Faced with the bottlenecks in our NFS-based design, we explored whether it would be useful to build a system similar to GFS [9]. Since we store most of our user data in MySQL databases, the main use cases for files in our system were the directories engineers use for development work, log data, and photos. NAS appliances offer a very good price/performance point for development work and for log data. Furthermore, we leverage Hadoop [11] for the extremely large log data. Serving photo requests in the long tail represents a problem for which neither MySQL, NAS appliances, nor Hadoop are well-suited.

One could phrase the dilemma we faced as existing storage systems lacked the right RAM-to-disk ratio. However, there is no *right* ratio. The system just needs *enough* main memory so that all of the filesystem metadata can be cached at once. In our NAS-based approach, one photo corresponds to one file and each file requires at least one inode, which is hundreds of bytes large. Having enough main memory in this approach is not cost-effective. To achieve a better price/performance point, we decided to build a custom storage system that reduces the amount of filesystem metadata per photo so that having enough main memory is dramatically more cost-effective than buying more NAS appliances.

3 Design & Implementation

Facebook uses a CDN to serve popular images and leverages Haystack to respond to photo requests in the long tail efficiently. When a web site has an I/O bottleneck serving static content the traditional solution is to use a CDN. The CDN shoulders enough of the burden so that the storage system can process the remaining tail. At Facebook a CDN would have to cache an unrea-

sonably large amount of the static content in order for traditional (and inexpensive) storage approaches not to be I/O bound.

Understanding that in the near future CDNs would not fully solve our problems, we designed Haystack to address the critical bottleneck in our NFS-based approach: disk operations. We accept that requests for less popular photos may require disk operations, but aim to limit the number of such operations to only the ones necessary for reading actual photo data. Haystack achieves this goal by dramatically reducing the memory used for filesystem metadata, thereby making it practical to keep all this metadata in main memory.

Recall that storing a single photo per file resulted in more filesystem metadata than could be reasonably cached. Haystack takes a straight-forward approach: it stores multiple photos in a single file and therefore maintains very large files. We show that this straightforward approach is remarkably effective. Moreover, we argue that its simplicity is its strength, facilitating rapid implementation and deployment. We now discuss how this core technique and the architectural components surrounding it provide a reliable and available storage system. In the following description of Haystack, we distinguish between two kinds of metadata. *Application metadata* describes the information needed to construct a URL that a browser can use to retrieve a photo. *Filesystem metadata* identifies the data necessary for a host to retrieve the photos that reside on that host's disk.

3.1 Overview

The Haystack architecture consists of 3 core components: the Haystack Store, Haystack Directory, and Haystack Cache. For brevity we refer to these components with 'Haystack' elided. The Store encapsulates the persistent storage system for photos and is the only component that manages the filesystem metadata for photos. We organize the Store's capacity by *physical volumes*. For example, we can organize a server's 10 terabytes of capacity into 100 physical volumes each of which provides 100 gigabytes of storage. We further group physical volumes on different machines into *logical volumes*. When Haystack stores a photo on a logical volume, the photo is written to all corresponding physical volumes. This redundancy allows us to mitigate data loss due to hard drive failures, disk controller bugs, etc. The Directory maintains the logical to physical mapping along with other application metadata, such as the logical volume where each photo resides and the logical volumes with free space. The Cache functions as our internal CDN, which shelters the Store from requests for the most popular photos and provides insulation if upstream CDN nodes fail and need to refetch content.

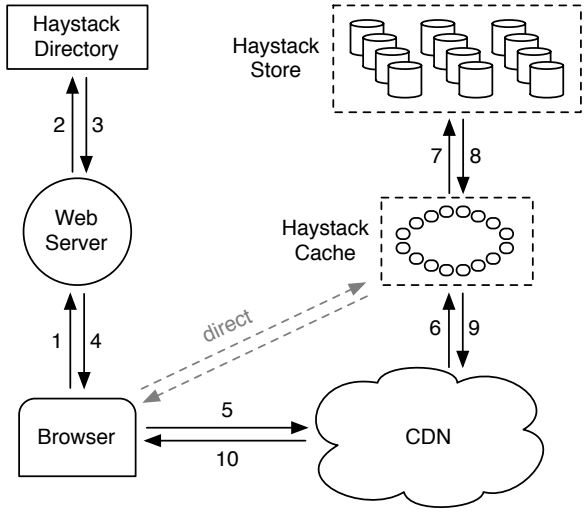


Figure 3: Serving a photo

Figure 3 illustrates how the Store, Directory, and Cache components fit into the canonical interactions between a user's browser, web server, CDN, and storage system. In the Haystack architecture the browser can be directed to either the CDN or the Cache. Note that while the Cache is essentially a CDN, to avoid confusion we use 'CDN' to refer to external systems and 'Cache' to refer to our internal one that caches photos. Having an internal caching infrastructure gives us the ability to reduce our dependence on external CDNs.

When a user visits a page the web server uses the Directory to construct a URL for each photo. The URL contains several pieces of information, each piece corresponding to the sequence of steps from when a user's browser contacts the CDN (or Cache) to ultimately retrieving a photo from a machine in the Store. A typical URL that directs the browser to the CDN looks like the following:

http://⟨CDN⟩/⟨Cache⟩/⟨Machine id⟩/⟨Logical volume, Photo⟩

The first part of the URL specifies from which CDN to request the photo. The CDN can lookup the photo internally using only the last part of the URL: the logical volume and the photo id. If the CDN cannot locate the photo then it strips the CDN address from the URL and contacts the Cache. The Cache does a similar lookup to find the photo and, on a miss, strips the Cache address from the URL and requests the photo from the specified Store machine. Photo requests that go directly to the Cache have a similar workflow except that the URL is missing the CDN specific information.

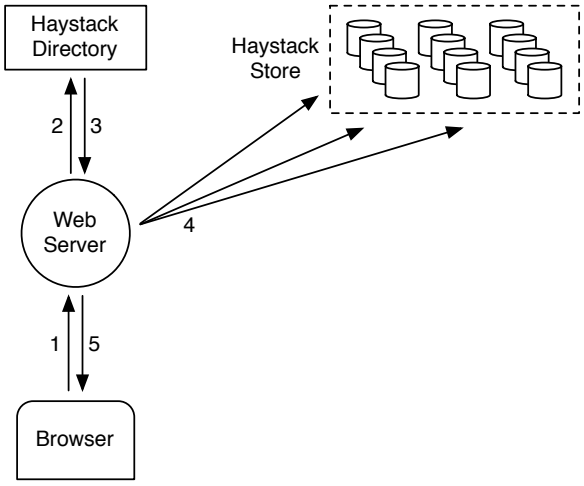


Figure 4: Uploading a photo

Figure 4 illustrates the upload path in Haystack. When a user uploads a photo she first sends the data to a web server. Next, that server requests a write-enabled logical volume from the Directory. Finally, the web server assigns a unique id to the photo and uploads it to each of the physical volumes mapped to the assigned logical volume.

3.2 Haystack Directory

The Directory serves four main functions. First, it provides a mapping from logical volumes to physical volumes. Web servers use this mapping when uploading photos and also when constructing the image URLs for a page request. Second, the Directory load balances writes across logical volumes and reads across physical volumes. Third, the Directory determines whether a photo request should be handled by the CDN or by the Cache. This functionality lets us adjust our dependence on CDNs. Fourth, the Directory identifies those logical volumes that are read-only either because of operational reasons or because those volumes have reached their storage capacity. We mark volumes as read-only at the granularity of machines for operational ease.

When we increase the capacity of the Store by adding new machines, those machines are write-enabled; only write-enabled machines receive uploads. Over time the available capacity on these machines decreases. When a machine exhausts its capacity, we mark it as read-only. In the next subsection we discuss how this distinction has subtle consequences for the Cache and Store.

The Directory is a relatively straight-forward component that stores its information in a replicated database accessed via a PHP interface that leverages memcache to reduce latency. In the event that we lose the data on a Store machine we remove the corresponding entry in the mapping and replace it when a new Store machine is brought online.

3.3 Haystack Cache

The Cache receives HTTP requests for photos from CDNs and also directly from users' browsers. We organize the Cache as a distributed hash table and use a photo's id as the key to locate cached data. If the Cache cannot immediately respond to the request, then the Cache fetches the photo from the Store machine identified in the URL and replies to either the CDN or the user's browser as appropriate.

We now highlight an important behavioral aspect of the Cache. It caches a photo only if two conditions are met: *(a)* the request comes directly from a user and not the CDN and *(b)* the photo is fetched from a write-enabled Store machine. The justification for the first condition is that our experience with the NFS-based design showed post-CDN caching is ineffective as it is unlikely that a request that misses in the CDN would hit in our internal cache. The reasoning for the second is indirect. We use the Cache to shelter write-enabled Store machines from reads because of two interesting properties: photos are most heavily accessed soon after they are uploaded and filesystems for our workload generally perform better when doing either reads or writes but not both (Section 4.1). Thus the write-enabled Store machines would see the most reads if it were not for the Cache. Given this characteristic, an optimization we plan to implement is to proactively push recently uploaded photos into the Cache as we expect those photos to be read soon and often.

3.4 Haystack Store

The interface to Store machines is intentionally basic. Reads make very specific and well-contained requests asking for a photo with a given id, for a certain logical volume, and from a particular physical Store machine. The machine returns the photo if it is found. Otherwise, the machine returns an error.

Each Store machine manages multiple physical volumes. Each volume holds millions of photos. For concreteness, the reader can think of a physical volume as simply a very large file (100 GB) saved as '/hay/haystack_<logical volume id>'. A Store machine can access a photo quickly using only the id of the corresponding logical volume and the file offset at which the photo resides. This knowledge is the keystone of the Haystack design: retrieving the filename, offset, and size for a particular photo without needing disk operations. A Store machine keeps open file descriptors for

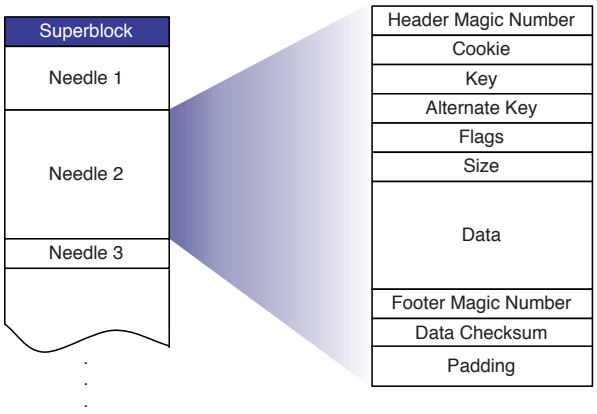


Figure 5: Layout of Haystack Store file

Field	Explanation
Header	Magic number used for recovery
Cookie	Random number to mitigate brute force lookups
Key	64-bit photo id
Alternate key	32-bit supplemental id
Flags	Signifies deleted status
Size	Data size
Data	The actual photo data
Footer	Magic number for recovery
Data Checksum	Used to check integrity
Padding	Total needle size is aligned to 8 bytes

Table 1: Explanation of fields in a needle

each physical volume that it manages and also an in-memory mapping of photo ids to the filesystem metadata (i.e., file, offset and size in bytes) critical for retrieving that photo.

We now describe the layout of each physical volume and how to derive the in-memory mapping from that volume. A Store machine represents a physical volume as a large file consisting of a superblock followed by a sequence of *needles*. Each needle represents a photo stored in Haystack. Figure 5 illustrates a volume file and the format of each needle. Table 1 describes the fields in each needle.

To retrieve needles quickly, each Store machine maintains an in-memory data structure for each of its volumes. That data structure maps pairs of (key, alternate key)[2] to the corresponding needle's flags, size in bytes, and volume offset. After a crash, a Store machine can reconstruct this mapping directly from the volume file before processing requests. We now describe how a Store machine maintains its volumes and in-memory mapping while responding to read, write, and delete requests (the only operations supported by the Store).

3.4.1 Photo Read

When a Cache machine requests a photo it supplies the logical volume id, key, alternate key, and cookie to the Store machine. The cookie is a number embedded in the URL for a photo. The cookie's value is randomly assigned by and stored in the Directory at the time that the photo is uploaded. The cookie effectively eliminates attacks aimed at guessing valid URLs for photos.

When a Store machine receives a photo request from a Cache machine, the Store machine looks up the relevant metadata in its in-memory mappings. If the photo has not been deleted the Store machine seeks to the appropriate offset in the volume file, reads the entire needle from disk (whose size it can calculate ahead of time), and verifies the cookie and the integrity of the data. If these checks pass then the Store machine returns the photo to the Cache machine.

3.4.2 Photo Write

When uploading a photo into Haystack web servers provide the logical volume id, key, alternate key, cookie, and data to Store machines. Each machine synchronously appends needle images to its physical volume files and updates in-memory mappings as needed. While simple, this append-only restriction complicates some operations that modify photos, such as rotations. As Haystack disallows overwriting needles, photos can only be modified by adding an updated needle with the same key and alternate key. If the new needle is written to a different logical volume than the original, the Directory updates its application metadata and future requests will never fetch the older version. If the new needle is written to the same logical volume, then Store machines append the new needle to the same corresponding physical volumes. Haystack distinguishes such duplicate needles based on their offsets. That is, the latest version of a needle within a physical volume is the one at the highest offset.

3.4.3 Photo Delete

Deleting a photo is straight-forward. A Store machine sets the delete flag in both the in-memory mapping and synchronously in the volume file. Requests to get deleted photos first check the in-memory flag and return errors if that flag is enabled. Note that the space occu-

[2]For historical reasons, a photo's id corresponds to the key while its type is used for the alternate key. During an upload, web servers scale each photo to four different sizes (or types) and store them as separate needles, but with the same key. The important distinction among these needles is the alternate key field, which in decreasing order can be 'n,' 'a,' 's,' or 't'.

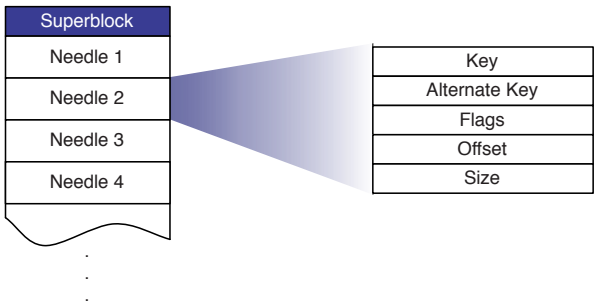


Figure 6: Layout of Haystack Index file

Field	Explanation
Key	64-bit key
Alternate key	32-bit alternate key
Flags	Currently unused
Offset	Needle offset in the Haystack Store
Size	Needle data size

Table 2: Explanation of fields in index file.

pied by deleted needles is for the moment lost. Later, we discuss how to reclaim deleted needle space by compacting volume files.

3.4.4 The Index File

Store machines use an important optimization—the *index file*—when rebooting. While in theory a machine can reconstruct its in-memory mappings by reading all of its physical volumes, doing so is time-consuming as the amount of data (terabytes worth) has to all be read from disk. Index files allow a Store machine to build its in-memory mappings quickly, shortening restart time.

Store machines maintain an index file for each of their volumes. The index file is a checkpoint of the in-memory data structures used to locate needles efficiently on disk. An index file's layout is similar to a volume file's, containing a superblock followed by a sequence of index records corresponding to each needle in the superblock. These records must appear in the same order as the corresponding needles appear in the volume file. Figure 6 illustrates the layout of the index file and Table 2 explains the different fields in each record.

Restarting using the index is slightly more complicated than just reading the indices and initializing the in-memory mappings. The complications arise because index files are updated asynchronously, meaning that index files may represent stale checkpoints. When we write a new photo the Store machine synchronously appends a needle to the end of the volume file and asynchronously appends a record to the index file. When we delete a photo, the Store machine synchronously sets the flag in that photo's needle without updating the index file. These design decisions allow write and delete operations to return faster because they avoid additional synchronous disk writes. They also cause two side effects we must address: needles can exist without corresponding index records and index records do not reflect deleted photos.

We refer to needles without corresponding index records as *orphans*. During restarts, a Store machine sequentially examines each orphan, creates a matching index record, and appends that record to the index file. Note that we can quickly identify orphans because the last record in the index file corresponds to the last non-orphan needle in the volume file. To complete the restart, the Store machine now initializes its in-memory mappings using only the index files.

Since index records do not reflect deleted photos, a Store machine may retrieve a photo that has in fact been deleted. To address this issue, after a Store machine reads the entire needle for a photo, that machine can then inspect the deleted flag. If a needle is marked as deleted the Store machine updates its in-memory mapping accordingly and notifies the Cache that the object was not found.

3.4.5 Filesystem

We describe Haystack as an object store that utilizes a generic Unix-like filesystem, but some filesystems are better suited for Haystack than others. In particular, the Store machines should use a filesystem that does not need much memory to be able to perform random seeks within a large file quickly. Currently, each Store machine uses XFS [24], an extent based file system. XFS has two main advantages for Haystack. First, the blockmaps for several contiguous large files can be small enough to be stored in main memory. Second, XFS provides efficient file preallocation, mitigating fragmentation and reining in how large block maps can grow.

Using XFS, Haystack can eliminate disk operations for retrieving filesystem metadata when reading a photo. This benefit, however, does not imply that Haystack can *guarantee* every photo read will incur exactly one disk operation. There exists corner cases where the filesystem requires more than one disk operation when photo data crosses extents or RAID boundaries. Haystack preallocates 1 gigabyte extents and uses 256 kilobyte RAID stripe sizes so that in practice we encounter these cases rarely.

3.5 Recovery from failures

Like many other large-scale systems running on commodity hardware [5, 4, 9], Haystack needs to tolerate a variety of failures: faulty hard drives, misbehaving RAID controllers, bad motherboards, etc. We use two straight-forward techniques to tolerate failures—one for detection and another for repair.

To proactively find Store machines that are having problems, we maintain a background task, dubbed *pitchfork*, that periodically checks the health of each Store machine. Pitchfork remotely tests the connection to each Store machine, checks the availability of each volume file, and attempts to read data from the Store machine. If pitchfork determines that a Store machine consistently fails these health checks then pitchfork automatically marks all logical volumes that reside on that Store machine as read-only. We manually address the underlying cause for the failed checks offline.

Once diagnosed, we may be able to fix the problem quickly. Occasionally, the situation requires a more heavy-handed *bulk sync* operation in which we reset the data of a Store machine using the volume files supplied by a replica. Bulk syncs happen rarely (a few each month) and are simple albeit slow to carry out. The main bottleneck is that the amount of data to be bulk synced is often orders of magnitude greater than the speed of the NIC on each Store machine, resulting in hours for mean time to recovery. We are actively exploring techniques to address this constraint.

3.6 Optimizations

We now discuss several optimizations important to Haystack's success.

3.6.1 Compaction

Compaction is an online operation that reclaims the space used by deleted and duplicate needles (needles with the same key and alternate key). A Store machine compacts a volume file by copying needles into a new file while skipping any duplicate or deleted entries. During compaction, deletes go to both files. Once this procedure reaches the end of the file, it blocks any further modifications to the volume and atomically swaps the files and in-memory structures.

We use compaction to free up space from deleted photos. The pattern for deletes is similar to photo views: young photos are a lot more likely to be deleted. Over the course of a year, about 25% of the photos get deleted.

3.6.2 Saving more memory

As described, a Store machine maintains an in-memory data structure that includes flags, but our current system only uses the flags field to mark a needle as deleted. We eliminate the need for an in-memory representation of flags by setting the offset to be 0 for deleted photos. In addition, Store machines do not keep track of cookie values in main memory and instead check the supplied cookie after reading a needle from disk. Store machines reduce their main memory footprints by 20% through these two techniques.

Currently, Haystack uses on average 10 bytes of main memory per photo. Recall that we scale each uploaded image to four photos all with the same key (64 bits), different alternate keys (32 bits), and consequently different data sizes (16 bits). In addition to these 32 bytes, Haystack consumes approximately 2 bytes per image in overheads due to hash tables, bringing the total for four scaled photos of the same image to 40 bytes. For comparison, consider that an `xfs_inode_t` structure in Linux is 536 bytes.

3.6.3 Batch upload

Since disks are generally better at performing large sequential writes instead of small random writes, we batch uploads together when possible. Fortunately, many users upload entire albums to Facebook instead of single pictures, providing an obvious opportunity to batch the photos in an album together. We quantify the improvement of aggregating writes together in Section 4.

4 Evaluation

We divide our evaluation into four parts. In the first we characterize the photo requests seen by Facebook. In the second and third we show the effectiveness of the Directory and Cache, respectively. In the last we analyze how well the Store performs using both synthetic and production workloads.

4.1 Characterizing photo requests

Photos are one of the primary kinds of content that users share on Facebook. Users upload millions of photos every day and recently uploaded photos tend to be much more popular than older ones. Figure 7 illustrates how popular each photo is as a function of the photo's age. To understand the shape of the graph, it is useful to discuss what drives Facebook's photo requests.

4.1.1 Features that drive photo requests

Two features are responsible for 98% of Facebook's photo requests: News Feed and albums. The News Feed feature shows users recent content that their friends have shared. The album feature lets a user browse her friends' pictures. She can view recently uploaded photos and also browse all of the individual albums.

Figure 7 shows a sharp rise in requests for photos that are a few days old. News Feed drives much of the traffic for recent photos and falls sharply away around 2 days when many stories stop being shown in the default Feed

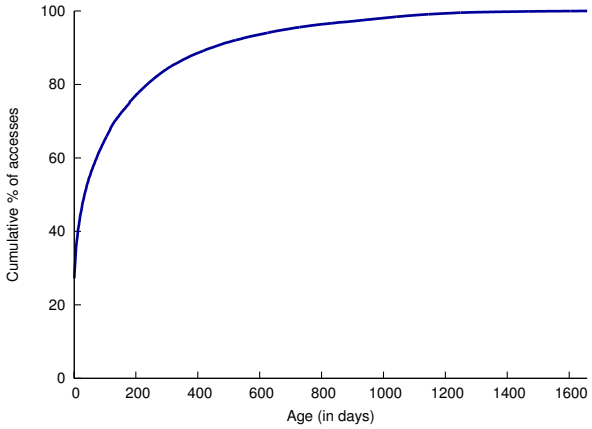


Figure 7: Cumulative distribution function of the number of photos requested in a day categorized by age (time since it was uploaded).

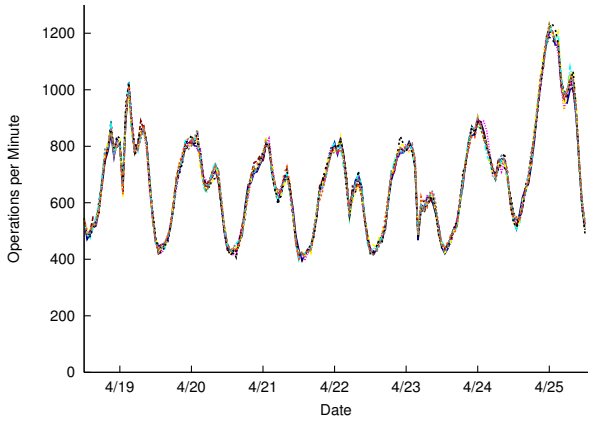


Figure 8: Volume of multi-write operations sent to 9 different write-enabled Haystack Store machines. The graph has 9 different lines that closely overlap each other.

Operations	**Daily Counts**
Photos Uploaded	~120 Million
Haystack Photos Written	~1.44 Billion
Photos Viewed	80-100 Billion
[*Thumbnails*]	10.2 %
[*Small*]	84.4 %
[*Medium*]	0.2 %
[*Large*]	5.2 %
Haystack Photos Read	10 Billion

Table 3: Volume of daily photo traffic.

view. There are two key points to highlght from the figure. First, the rapid decline in popularity suggests that caching at both CDNs and in the Cache can be very effective for hosting popular content. Second, the graph has a long tail implying that a significant number of requests cannot be dealt with using cached data.

4.1.2 Traffic Volume

Table 3 shows the volume of photo traffic on Facebook. The number of Haystack photos written is 12 times the number of photos uploaded since our application scales each image to 4 sizes and saves each size in 3 different locations. The table shows that Haystack responds to approximately 10% of all photo requests from CDNs. Observe that smaller images account for most of the photos viewed. This trait underscores our desire to minimize metadata overhead as inefficiencies can quickly add up. Additionally, reading smaller images is typically a more latency sensitive operation for Facebook as they are displayed in the News Feed whereas larger images are shown in albums and can be prefetched to hide latency.

4.2 Haystack Directory

The Haystack Directory balances reads and writes across Haystack Store machines. Figure 8 depicts that as expected, the Directory's straight-forward hashing policy to distribute reads and writes is very effective. The graph shows the number of multi-write operations seen by 9 different Store machines which were deployed into production at the same time. Each of these boxes store a different set of photos. Since the lines are nearly indistinguishable, we conclude that the Directory balances writes well. Comparing read traffic across Store machines shows similarly well-balanced behavior.

4.3 Haystack Cache

Figure 9 shows the hit rate for the Haystack Cache. Recall that the Cache only stores a photo if it is saved on a write-enabled Store machine. These photos are relatively recent, which explains the high hit rates of approximately 80%. Since the write-enabled Store machines would also see the greatest number of reads, the Cache is effective in dramatically reducing the read request rate for the machines that would be most affected.

4.4 Haystack Store

Recall that Haystack targets the long tail of photo requests and aims to maintain high-throughput and low-latency despite seemingly random reads. We present performance results of Store machines on both synthetic and production workloads.

Benchmark	[Config # Operations]	Reads			Writes		
		Throughput (in images/s)	Latency (in ms)		Throughput (in images/s)	Latency (in ms)	
			Avg.	Std. dev.		Avg.	Std. dev.
Random IO	[Only Reads]	902.3	33.2	26.8	–	–	–
Haystress	[**A** # Only Reads]	770.6	38.9	30.2	–	–	–
Haystress	[**B** # Only Reads]	877.8	34.2	28.1	–	–	–
Haystress	[**C** # Only Multi-Writes]	–	–	–	6099.4	4.9	16.0
Haystress	[**D** # Only Multi-Writes]	–	–	–	7899.7	15.2	15.3
Haystress	[**E** # Only Multi-Writes]	–	–	–	10843.8	43.9	16.3
Haystress	[**F** # Reads & Multi-Writes]	718.1	41.6	31.6	232.0	11.9	6.3
Haystress	[**G** # Reads & Multi-Writes]	692.8	42.8	33.7	440.0	11.9	6.9

Table 4: Throughput and latency of read and multi-write operations on synthetic workloads. Config **B** uses a mix of 8KB and 64KB images. Remaining configs use 64KB images.

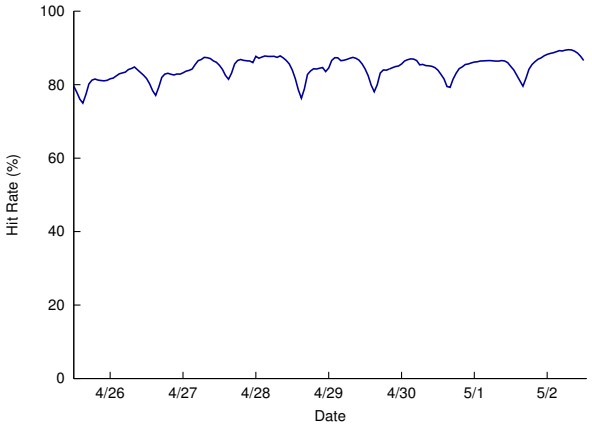


Figure 9: Cache hit rate for images that might be potentially stored in the Haystack Cache.

4.4.1 Experimental setup

We deploy Store machines on commodity storage blades. The typical hardware configuration of a 2U storage blade has 2 hyper-threaded quad-core Intel Xeon CPUs, 48 GB memory, a hardware raid controller with 256–512MB NVRAM, and 12 x 1TB SATA drives.

Each storage blade provides approximately 9TB of capacity, configured as a RAID-6 partition managed by the hardware RAID controller. RAID-6 provides adequate redundancy and excellent read performance while keeping storage costs down. The controller's NVRAM write-back cache mitigates RAID-6's reduced write performance. Since our experience suggests that caching photos on Store machines is ineffective, we reserve the NVRAM fully for writes. We also disable disk caches in order to guarantee data consistency in the event of a crash or power loss.

4.4.2 Benchmark performance

We assess the performance of a Store machine using two benchmarks: Randomio [22] and Haystress. Randomio is an open-source multithreaded disk I/O program that we use to measure the raw capabilities of storage devices. It issues random 64KB reads that use direct I/O to make sector aligned requests and reports the maximum sustainable throughput. We use Randomio to establish a baseline for read throughput against which we can compare results from our other benchmark.

Haystress is a custom built multi-threaded program that we use to evaluate Store machines for a variety of synthetic workloads. It communicates with a Store machine via HTTP (as the Cache would) and assesses the maximum read and write throughput a Store machine can maintain. Haystress issues random reads over a large set of dummy images to reduce the effect of the machine's buffer cache; that is, nearly all reads require a disk operation. In this paper, we use seven different Haystress workloads to evaluate Store machines.

Table 4 characterizes the read and write throughputs and associated latencies that a Store machine can sustain under our benchmarks. Workload **A** performs random reads to 64KB images on a Store machine with 201 volumes. The results show that Haystack delivers 85% of the raw throughput of the device while incurring only 17% higher latency.

We attribute a Store machine's overhead to four factors: (a) it runs on top of the filesystem instead of accessing disk directly; (b) disk reads are larger than 64KB as entire needles need to be read; (c) stored images may not be aligned to the underlying RAID-6 device stripe size so a small percentage of images are read from more

than one disk; and (d) CPU overhead of Haystack server (index access, checksum calculations, etc.)

In workload **B**, we again examine a read-only workload but alter 70% of the reads so that they request smaller size images (8KB instead of 64KB). In practice, we find that most requests are not for the largest size images (as would be shown in albums) but rather for the thumbnails and profile pictures.

Workloads **C**, **D**, and **E** show a Store machine's write throughput. Recall that Haystack can batch writes together. Workloads **C**, **D**, and **E** group 1, 4, and 16 writes into a single multi-write, respectively. The table shows that amortizing the fixed cost of writes over 4 and 16 images improves throughput by 30% and 78% respectively. As expected, this reduces per image latency, as well.

Finally, we look at the performance in the presence of both read and write operations. Workload **F** uses a mix of 98% reads and 2% multi-writes while **G** uses a mix of 96% reads and 4% multi-writes where each multi-write writes 16 images. These ratios reflect what is often observed in production. The table shows that the Store delivers high read throughput even in the presence of writes.

4.4.3 Production workload

The section examines the performance of the Store on production machines. As noted in Section 3, there are two classes of Stores–write-enabled and read-only. Write-enabled hosts service read and write requests, read-only hosts only service read requests. Since these two classes have fairly different traffic characteristics, we analyze a group of machines in each class. All machines have the same hardware configuration.

Viewed at a per-second granularity, there can be large spikes in the volume of photo read and write operations that a Store box sees. To ensure reasonable latency even in the presence of these spikes, we conservatively allocate a large number of write-enabled machines so that their average utilization is low.

Figure 10 shows the frequency of the different types of operations on a read-only and a write-enabled Store machine. Note that we see peak photo uploads on Sunday and Monday, with a smooth drop the rest of the week until we level out on Thursday to Saturday. Then a new Sunday arrives and we hit a new weekly peak. In general our footprint grows by 0.2% to 0.5% per day.

As noted in Section 3, write operations to the Store are always multi-writes on production machines to amortize the fixed cost of write operations. Finding groups of images is fairly straightforward since 4 different sizes of each photo is stored in Haystack. It is also common for users to upload a batch of photos into

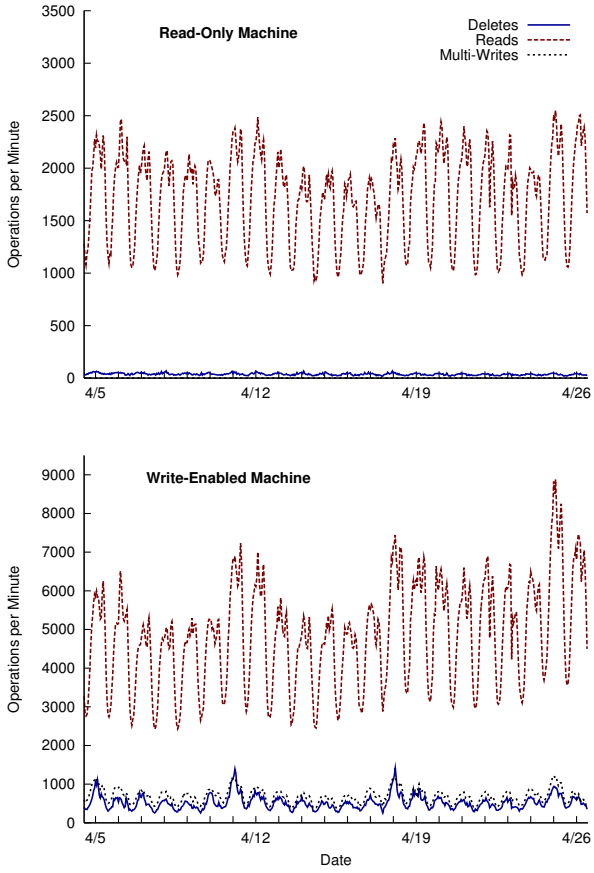


Figure 10: Rate of different operations on two Haystack Store machines: One read-only and the other write-enabled.

a photo album. As a combination of these two factors, the average number of images written per multi-write for this write-enabled machine is 9.27.

Section 4.1.2 explained that both read and delete rates are high for recently uploaded photos and drop over time. This behavior can be also be observed in Figure 10; the write-enabled boxes see many more requests (even though some of the read traffic is served by the Cache).

Another trend worth noting: as more data gets written to write-enabled boxes the volume of photos increases, resulting in an increase in the read request rate.

Figure 11 shows the latency of read and multi-write operations on the same two machines as Figure 10 over the same period.

The latency of multi-write operations is fairly low (between 1 and 2 milliseconds) and stable even as the volume of traffic varies dramatically. Haystack ma-

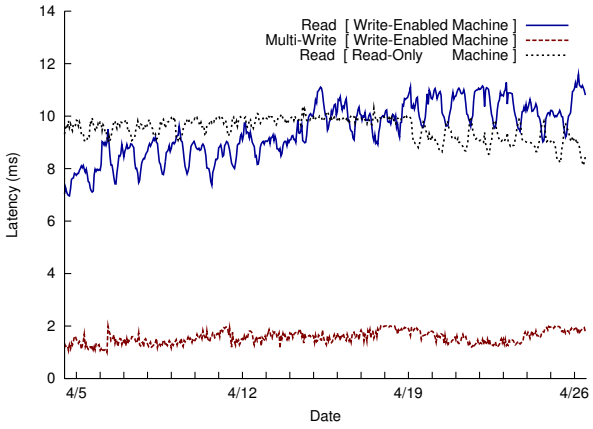


Figure 11: Average latency of Read and Multi-write operations on the two Haystack Store machines in Figure 10 over the same 3 week period.

chines have a NVRAM-backed raid controller which buffers writes for us. As described in Section 3, the NVRAM allows us to write needles asynchronously and then issue a single fsync to flush the volume file once the multi-write is complete. Multi-write latencies are very flat and stable.

The latency of reads on a read-only box is also fairly stable even as the volume of traffic varies significantly (up to 3x over the 3 week period). For a write-enabled box the read performance is impacted by three primary factors. First, as the number of photos stored on the machine increases, the read traffic to that machine also increases (compare week-over-week traffic in figure 10). Second, photos on write-enabled machines are cached in the Cache while they are not cached for a read-only machine[3]. This suggests that the buffer cache would be more effective for a read-only machine. Third, recently written photos are usually read back immediately because Facebook highlights recent content. Such reads on Write-enabled boxes will always hit in the buffer cache and improve the hit rate of the buffer cache. The shape of the line in the figure is the result of a combination of these three factors.

The CPU utilization on the Store machines is low. CPU idle time varies between 92-96%.

5 Related Work

To our knowledge, Haystack targets a new design point focusing on the long tail of photo requests seen by a large social networking website.

Filesystems Haystack takes after log-structured filesystems [23] which Rosenblum and Ousterhout designed to optimize write throughput with the idea that most reads could be served out of cache. While measurements [3] and simulations [6] have shown that log-structured filesystems have not reached their full potential in local filesystems, the core ideas are very relevant to Haystack. Photos are appended to physical volume files in the Haystack Store and the Haystack Cache shelters write-enabled machines from being overwhelmed by the request rate for recently uploaded data. The key differences are *(a)* that the Haystack Store machines write their data in such a way that they can efficiently serve reads once they become read-only and *(b)* the read request rate for older data decreases over time.

Several works [8, 19, 28] have proposed how to manage small files and metadata more efficiently. The common thread across these contributions is how to group related files and metadata together intelligently. Haystack obviates these problems since it maintains metadata in main memory and users often upload related photos in bulk.

Object-based storage Haystack's architecture shares many similarities with object storage systems proposed by Gibson et al. [10] in Network-Attached Secure Disks (NASD). The Haystack Directory and Store are perhaps most similar to the File and Storage Manager concepts, respectively, in NASD that separate the logical storage units from the physical ones. In OBFS [25], Wang et al. build a user-level object-based filesystem that is $\frac{1}{25^{th}}$ the size of XFS. Although OBFS achieves greater write throughput than XFS, its read throughput (Haystack's main concern) is slightly worse.

Managing metadata Weil et al. [26, 27] address scaling metadata management in Ceph, a petabyte-scale object store. Ceph further decouples the mapping from logical units to physical ones by introducing generating functions instead of explicit mappings. Clients can *calculate* the appropriate metadata rather than look it up. Implementing this technique in Haystack remains future work. Hendricks et. al [13] observe that traditional metadata pre-fetching algorithms are less effective for object stores because related objects, which are identified by a unique number, lack the semantic groupings that directories implicitly impose. Their solution is to embed inter-object relationships into the object id. This idea is orthogonal to Haystack as Facebook explicitly stores these semantic relationships as part of the social

[3]Note that for traffic coming through a CDN, they are cached in the CDNs and not in the Cache in both instances

graph. In Spyglass [15], Leung et al. propose a design for quickly and scalably searching through metadata of large-scale storage systems. Manber and Wu also propose a way to search through entire filesystems in GLIMPSE [17]. Patil et al. [20] use a sophisticated algorithm in GIGA+ to manage the metadata associated with billions of files per directory. We engineered a simpler solution than many existing works as Haystack does not have to provide search features nor traditional UNIX filesystem semantics.

Distributed filesystems Haystack's notion of a logical volume is similar to Lee and Thekkath's [14] *virtual disks* in Petal. The Boxwood project [16] explores using high-level data structures as the foundation for storage. While compelling for more complicated algorithms, abstractions like B-trees may not have high impact on Haystack's intentionally lean interface and semantics. Similarly, Sinfonia's [1] *mini-transactions* and PNUTS's [5] database functionality provide more features and stronger guarantees than Haystack needs. Ghemawat et al. [9] designed the Google File System for a workload consisting mostly of append operations and large sequential reads. Bigtable [4] provides a storage system for structured data and offers database-like features for many of Google's projects. It is unclear whether many of these features make sense in a system optimized for photo storage.

6 Conclusion

This paper describes Haystack, an object storage system designed for Facebook's Photos application. We designed Haystack to serve the long tail of requests seen by sharing photos in a large social network. The key insight is to avoid disk operations when accessing metadata. Haystack provides a fault-tolerant and simple solution to photo storage at dramatically less cost and higher throughput than a traditional approach using NAS appliances. Furthermore, Haystack is incrementally scalable, a necessary quality as our users upload hundreds of millions of photos each week.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 159–174, New York, NY, USA, 2007. ACM.

[2] Akamai. http://www.akamai.com/.

[3] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proc. 13th SOSP*, pages 198–212, 1991.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[5] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[6] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, Nov 1994.

[7] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: the future building block for storage systems. In *LGDI '05: Proceedings of the 2005 IEEE International Symposium on Mass Storage Systems and Technology*, pages 119–123, Washington, DC, USA, 2005. IEEE Computer Society.

[8] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. In *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 1997. USENIX Association.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. 19th SOSP*, pages 29–43. ACM Press, 2003.

[10] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. *SIGOPS Oper. Syst. Rev.*, 32(5):92–103, 1998.

[11] The hadoop project. http://hadoop.apache.org/.

[12] S. He and D. Feng. Design of an object-based storage device based on i/o processor. *SIGOPS Oper. Syst. Rev.*, 42(6):30–35, 2008.

[13] J. Hendricks, R. R. Sambasivan, S. Sinnamohideen, and G. R. Ganger. Improving small file performance in object-based storage. Technical Report 06-104, Parallel Data Laboratory, Carnegie Mellon University, 2006.

[14] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 84–92, New York, NY, USA, 1996. ACM.

[15] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller. Spyglass: fast, scalable metadata search for large-scale storage systems. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 153–166, Berkeley, CA, USA, 2009. USENIX Association.

[16] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.

[17] U. Manber and S. Wu. Glimpse: a tool to search through entire file systems. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 4–4, Berkeley, CA, USA, 1994. USENIX Association.

[18] memcache. http://memcached.org/.

[19] S. J. Mullender and A. S. Tanenbaum. Immediate files. *Softw. Pract. Exper.*, 14(4):365–368, 1984.

[20] S. V. Patil, G. A. Gibson, S. Lang, and M. Polte. Giga+: scalable directories for shared file systems. In *PDSW '07: Proceedings of the 2nd international workshop on Petascale data storage*, pages 26–29, New York, NY, USA, 2007. ACM.

[21] Posix. http://standards.ieee.org/regauth/posix/.

[22] Randomio. http://members.optusnet.com.au/clausen/ideas/randomio/index.html.

[23] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[24] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the xfs file system. In *ATEC '96: Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[25] F. Wang, S. A. Brandt, E. L. Miller, and D. D. E. Long. Obfs: A file system for object-based storage devices. In *In Proceedings of the 21st IEEE / 12TH NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 283–300, 2004.

[26] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[27] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 4, Washington, DC, USA, 2004. IEEE Computer Society.

[28] Z. Zhang and K. Ghose. hfs: a hybrid file system prototype for improving small file and metadata performance. *SIGOPS Oper. Syst. Rev.*, 41(3):175–187, 2007.

Availability in Globally Distributed Storage Systems

Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong*,
Luiz Barroso, Carrie Grimes, and Sean Quinlan
{ford,flab,florentina,mstokely}@google.com, vatruong@ieor.columbia.edu
{luiz,cgrimes,sean}@google.com
Google, Inc.

Abstract

Highly available cloud storage is often implemented with complex, multi-tiered distributed systems built on top of clusters of commodity servers and disk drives. Sophisticated management, load balancing and recovery techniques are needed to achieve high performance and availability amidst an abundance of failure sources that include software, hardware, network connectivity, and power issues. While there is a relative wealth of failure studies of individual components of storage systems, such as disk drives, relatively little has been reported so far on the overall availability behavior of large cloud-based storage services.

We characterize the availability properties of cloud storage systems based on an extensive one year study of Google's main storage infrastructure and present statistical models that enable further insight into the impact of multiple design choices, such as data placement and replication strategies. With these models we compare data availability under a variety of system parameters given the real patterns of failures observed in our fleet.

1 Introduction

Cloud storage is often implemented by complex multi-tiered distributed systems on clusters of thousands of commodity servers. For example, in Google we run Bigtable [9], on GFS [16], on local Linux file systems that ultimately write to local hard drives. Failures in any of these layers can cause data unavailability.

Correctly designing and optimizing these multi-layered systems for user goals such as data availability relies on accurate models of system behavior and performance. In the case of distributed storage systems, this includes quantifying the impact of failures and prioritizing hardware and software subsystem improvements in the datacenter environment.

We present models we derived from studying a year of live operation at Google and describe how our analysis influenced the design of our next generation distributed storage system [22].

Our work is presented in two parts. First, we measured and analyzed the *component availability*, e.g. machines, racks, multi-racks, in tens of Google storage clusters. In this part we:

- Compare mean time to failure for system components at different granularities, including disks, machines and racks of machines. (Section 3)
- Classify the failure causes for storage nodes, their characteristics and contribution to overall unavailability. (Section 3)
- Apply a clustering heuristic for grouping failures which occurs almost simultaneously and show that a large fraction of failures happen in bursts. (Section 4)
- Quantify how likely a failure burst is associated with a given failure domain. We find that most large bursts of failures are associated with rack- or multi-rack level events. (Section 4)

Based on these results, we determined that the critical element in models of availability is their ability to account for the frequency and magnitude of *correlated* failures.

Next, we consider *data availability* by analyzing unavailability at the distributed file system level, where one file system instance is referred to as a *cell*. We apply two models of multi-scale correlated failures for a variety of replication schemes and system parameters. In this part we:

- Demonstrate the importance of modeling correlated failures when predicting availability, and show their

*Now at Dept. of Industrial Engineering and Operations Research Columbia University

impact under a variety of replication schemes and placement policies. (Sections 5 and 6)

- Formulate a Markov model for data availability, that can scale to arbitrary cell sizes, and captures the interaction of failures with replication policies and recovery times. (Section 7)
- Introduce multi-cell replication schemes and compare the availability and bandwidth trade-offs against single-cell schemes. (Sections 7 and 8)
- Show the impact of hardware failure on our cells is significantly smaller than the impact of effectively tuning recovery and replication parameters. (Section 8)

Our results show the importance of considering cluster-wide failure events in the choice of replication and recovery policies.

2 Background

We study end to end data availability in a cloud computing storage environment. These environments often use loosely coupled distributed storage systems such as GFS [1, 16] due to the parallel I/O and cost advantages they provide over traditional SAN and NAS solutions. A few relevant characteristics of such systems are:

- Storage server programs running on physical machines in a datacenter, managing local disk storage on behalf of the distributed storage cluster. We refer to the storage server programs as *storage nodes* or *nodes*.
- A pool of storage service masters managing data placement, load balancing and recovery, and monitoring of storage nodes.
- A replication or erasure code mechanism for user data to provide resilience to individual component failures.

A large collection of nodes along with their higher level coordination processes [17] are called a *cell* or *storage cell*. These systems usually operate in a shared pool of machines running a wide variety of applications. A typical cell may comprise many thousands of nodes housed together in a single building or set of colocated buildings.

2.1 Availability

A storage node becomes *unavailable* when it fails to respond positively to periodic health checking pings sent by our monitoring system. The node remains unavailable until it regains responsiveness or the storage system reconstructs the data from other surviving nodes.

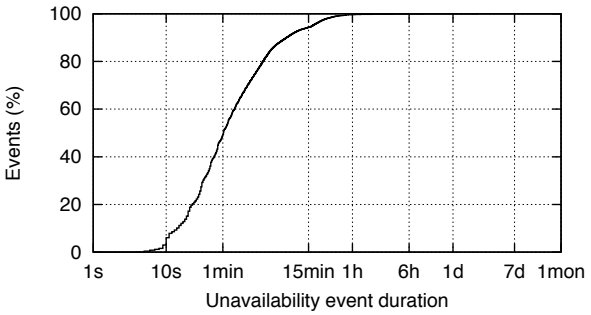


Figure 1: Cumulative distribution function of the duration of node unavailability periods.

Nodes can become unavailable for a large number of reasons. For example, a storage node or networking switch can be overloaded; a node binary or operating system may crash or restart; a machine may experience a hardware error; automated repair processes may temporarily remove disks or machines; or the whole cluster could be brought down for maintenance. The vast majority of such unavailability events are transient and do not result in permanent data loss. Figure 1 plots the CDF of node unavailability duration, showing that less than 10% of events last longer than 15 minutes. This data is gathered from tens of Google storage cells, each with 1000 to 7000 nodes, over a one year period. The cells are located in different datacenters and geographical regions, and have been used continuously by different projects within Google. We use this dataset throughout the paper, unless otherwise specified.

Experience shows that while short unavailability events are most frequent, they tend to have a minor impact on cluster-level availability and data loss. This is because our distributed storage systems typically add enough redundancy to allow data to be served from other sources when a particular node is unavailable. Longer unavailability events, on the other hand, make it more likely that faults will overlap in such a way that data could become unavailable at the cluster level for long periods of time. Therefore, while we track unavailability metrics at multiple time scales in our system, in this paper we focus only on events that are 15 minutes or longer. This interval is long enough to exclude the majority of benign transient events while not too long to exclude significant cluster-wide phenomena. As in [11], we observe that initiating recovery after transient failures is inefficient and reduces resources available for other operations. For these reasons, GFS typically waits 15 minutes before commencing recovery of data on unavailable nodes.

We primarily use two metrics throughout this paper. The average availability of all N nodes in a cell is defined as:

$$A_N = \frac{\sum_{N_i \in N} \text{uptime}(N_i)}{\sum_{N_i \in N} (\text{uptime}(N_i) + \text{downtime}(N_i))} \quad (1)$$

We use uptime(N_i) and downtime(N_i) to refer to the lengths of time a node N_i is available or unavailable, respectively. The sum of availability periods over all nodes is called *node uptime*. We define uptime similarly for other component types. We define unavailability as the complement of availability.

Mean time to failure, or *MTTF*, is commonly quoted in the literature related to the measurements of availability. We use MTTF for components that suffer transient or permanent failures, to avoid frequent switches in terminology.

$$MTTF = \frac{\text{uptime}}{\text{number failures}} \quad (2)$$

Availability measurements for nodes and individual components in our system are presented in Section 3.

2.2 Data replication

Distributed storage systems increase resilience to failures by using replication [2] or erasure encoding across nodes [28]. In both cases, data is divided into a set of *stripes*, each of which comprises a set of fixed size data and code blocks called *chunks*. Data in a stripe can be reconstructed from some subsets of the chunks. For replication, $R = n$ refers to n identical chunks in a stripe, so the data may be recovered from any one chunk. For Reed-Solomon erasure encoding, $RS(n, m)$ denotes n distinct data blocks and m error correcting blocks in each stripe. In this case a stripe may be reconstructed from any n chunks.

We call a chunk available if the node it is stored on is available. We call a stripe available if enough of its chunks are available to reconstruct the missing chunks, if any.

Data availability is a complex function of the individual node availability, the encoding scheme used, the distribution of correlated node failures, chunk placement, and recovery times that we will explore in the second part of this paper. We do not explore related mechanisms for dealing with failures, such as additional application level redundancy and recovery, and manual component repair.

3 Characterizing Node Availability

Anything that renders a storage node unresponsive is a potential cause of unavailability, including hardware component failures, software bugs, crashes, system reboots, power loss events, and loss of network connectivity. We include in our analysis the impact of software upgrades, reconfiguration, and other maintenance. These planned outages are necessary in a fast evolving datacenter environment, but have often been overlooked in other availability studies. In this section we present data for storage node unavailability and provide some insight into the main causes for unavailability.

3.1 Numbers from the fleet

Failure patterns vary dramatically across different hardware platforms, datacenter operating environments, and workloads. We start by presenting numbers for disks.

Disks have been the focus of several other studies, since they are the system component that permanently stores the data, and thus a disk failure potentially results in permanent data loss. The numbers we observe for disk and storage subsystem failures, presented in Table 2, are comparable with what other researchers have measured. One study [29] reports ARR (annual replacement rate) for disks between 2% and 4%. Another study [19] focused on storage subsystems, thus including errors from shelves, enclosures, physical interconnects, protocol failures, and performance failures. They found AFR (annual failure rate) generally between 2% and 4%, but for some storage systems values ranging between 3.9% and 8.3%.

For the purposes of this paper, we are interested in disk errors as perceived by the application layer. This includes latent sector errors and corrupt sectors on disks, as well as errors caused by firmware, device drivers, controllers, cables, enclosures, silent network and memory corruption, and software bugs. We deal with these errors with background scrubbing processes on each node, as in [5, 31], and by verifying data integrity during client reads [4]. Background scrubbing in GFS finds between 1 in 10^6 to 10^7 of older data blocks do not match the checksums recorded when the data was originally written. However, these cell-wide rates are typically concentrated on a small number of disks.

We are also concerned with node failures in addition to individual disk failures. Figure 2 shows the distribution of three mutually exclusive causes of node unavailability in one of our storage cells. We focus on *node restarts* (software restarts of the storage program running on each machine), *planned machine reboots* (e.g. kernel version upgrades), and *unplanned machine reboots* (e.g. kernel crashes). For the purposes of this figure we do not exclude events that last less than 15 minutes, but we still end the unavailability period when the system reconstructs all the data previously stored on that node. Node restart events exhibit the greatest variability in duration, ranging from less than one minute to well over an

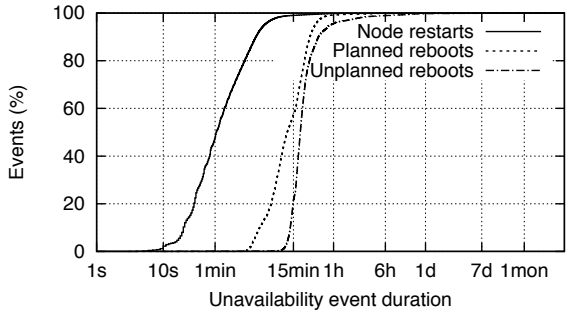


Figure 2: Cumulative distribution function of node unavailability durations by cause.

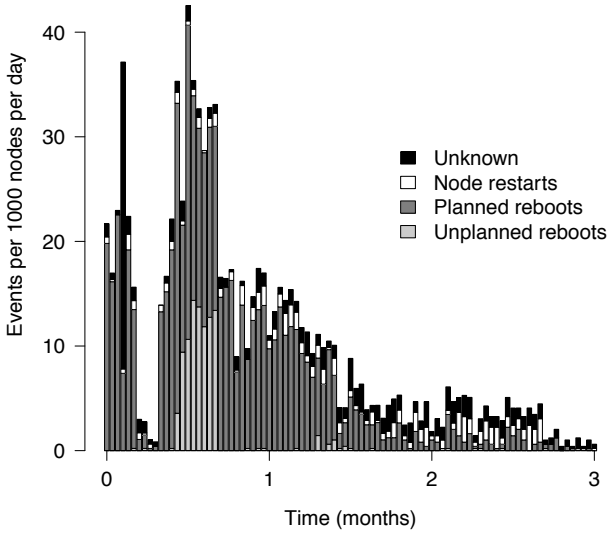


Figure 3: Rate of events per 1000 nodes per day, for one example cell.

hour, though they usually have the shortest duration. Unplanned reboots have the longest average duration since extra checks or corrective action is often required to restore machines to a safe state.

Figure 3 plots the unavailability events per 1000 nodes per day for one example cell, over a period of three months. The number of events per day, as well as the number of events that can be attributed to a given cause vary significantly over time as operational processes, tools, and workloads evolve. Events we cannot classify accurately are labeled *unknown*.

The effect of machine failures on availability is dependent on the rate of failures, as well as on how long the machines stay unavailable. Figure 4 shows the node unavailability, along with the causes that generated the unavailability, for the same cell used in Figure 3. The availability is computed with a one week rolling window, using definition (1). We observe that the majority of unavailability is generated by planned reboots.

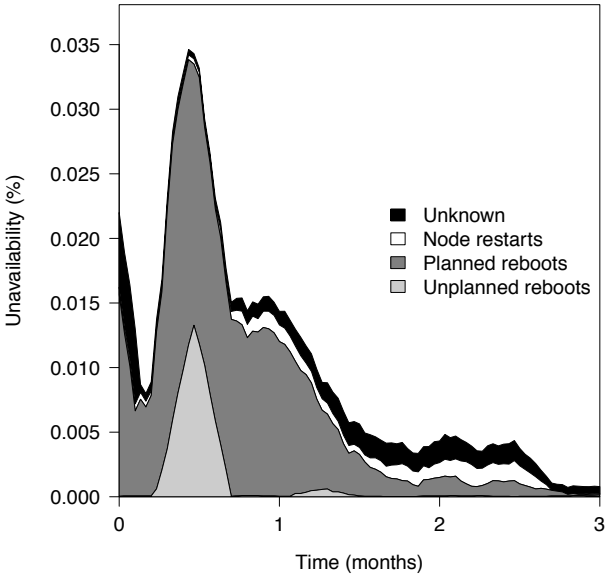


Figure 4: Storage node unavailability computed with a one week rolling window, for one example cell.

Cause	Unavailability (%) average / min / max
Node restarts	0.0139 / 0.0004 / 0.1295
Planned machine reboots	0.0154 / 0.0050 / 0.0563
Unplanned machine reboots	0.0025 / 0.0000 / 0.0122
Unknown	0.0142 / 0.0013 / 0.0454

Table 1: Unavailability attributed to different failure causes, over the full set of cells.

Table 1 shows the unavailability from node restarts, planned and unplanned machine reboots, each of which is a significant cause. The numbers are exclusive, thus the planned machine reboots do not include node restarts.

Table 2 shows the MTTF for a series of important components: disk, nodes, and racks of nodes. The numbers we report for component failures are inclusive of software errors and hardware failures. Though disks failures are permanent and most node failures are transitory, the significantly greater frequency of node failures makes them a much more important factor for system availability (Section 8.4).

4 Correlated Failures

The co-occurring failure of a large number of nodes can reduce the effectiveness of replication and encoding schemes. Therefore it is critical to take into account the statistical behavior of correlated failures to understand data availability. In this section we are more concerned with measuring the frequency and severity of such failures rather than root causes.

Component	Disk	Node	Rack
MTTF	10-50 years	4.3 months	10.2 years

Table 2: Component failures across several Google cells.

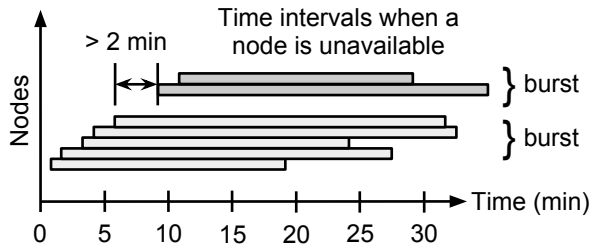


Figure 5: Seven node failures clustered into two failure bursts when the window size is 2 minutes. Note how only the unavailability start times matter.

We define a *failure burst* and examine features of these bursts in the field. We also develop a method for identifying which bursts are likely due to a failure domain. By failure domain, we mean a set of machines which we expect to simultaneously suffer from a common source of failure, such as machines which share a network switch or power cable. We demonstrate this method by validating physical racks as an important failure domain.

4.1 Defining failure bursts

We define a *failure burst* with respect to a window size w as a maximal sequence of node failures, each one occurring within a time window w of the next. Figure 5 illustrates the definition. We choose $w = 120$ s, for several reasons. First, it is longer than the frequency with which nodes are periodically polled in our system for their status. A window length smaller than the polling interval would not make sense as some pairs of events which actually occur within the window length of each other would not be correctly associated. Second, it is less than a tenth of the average time it takes our system to recover a chunk, thus, failures within this window can be considered as nearly concurrent. Figure 6 shows the fraction of individual failures that get clustered into bursts of at least 10 nodes as the window size changes. Note that the graph is relatively flat after 120 s, which is our third reason for choosing this value.

Since failures are clustered into bursts based on their times of occurrence alone, there is a risk that two bursts with independent causes will be clustered into a single burst by chance. The slow increase in Figure 6 past 120 s illustrates this phenomenon. The error incurred is small as long as we keep the window size small. Given a window size of 120 s and the set of bursts obtained from it, the probability that a random failure gets included in a burst (as opposed to becoming its own singleton burst) is 8.0%. When this inclusion happens, most of the time the random failure is combined with a singleton burst to form a burst of two nodes. The probability that a random failure gets included in a burst of at least 10 nodes is only 0.068%. For large bursts, which contribute most unavailability as we will see in Section 5.2, the fraction of nodes affected is the significant quantity and changes insignificantly if a burst of size one or two nodes is accidentally clustered with it.

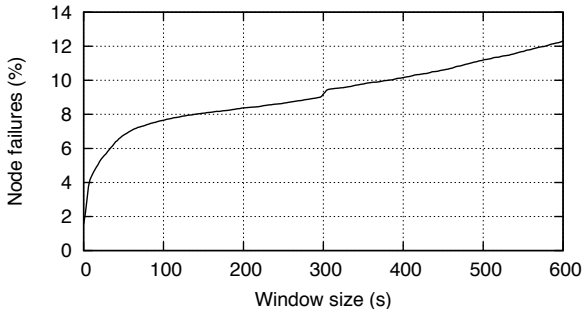


Figure 6: Effect of the window size on the fraction of individual failures that get clustered into bursts of at least 10 nodes.

Using this definition, we observe that 37% of failures are part of a burst of at least 2 nodes. Given the result above that only 8.0% of non-correlated failures may be incorrectly clustered, we are confident that close to 37% of failures are truly correlated.

4.2 Views of failure bursts

Figure 7 shows the accumulation of individual failures in bursts. For clarity we show all bursts of size at least 10 seen over a 60 day period in an example cell. In the plot, each burst is displayed with a separate shape. The n-th node failure that joins a burst at time t_n is said to have ordinal $n - 1$ and is plotted at point $(t_n, n - 1)$. Two broad classes of failure bursts can be seen in the plot:

1. Those failure bursts that are characterized by a large number of failures in quick succession show up as steep lines with a large number of nodes in the burst. Such failures can be seen, for example, following a power outage in a datacenter.

2. Those failure bursts that are characterized by a smaller number of nodes failing at a slower rate at evenly spaced intervals. Such correlated failures can be seen, for example, as part of rolling reboot or upgrade activity at the datacenter management layer.

Figure 8 displays the bursts sorted by the number of nodes and racks that they affect. The size of each bubble

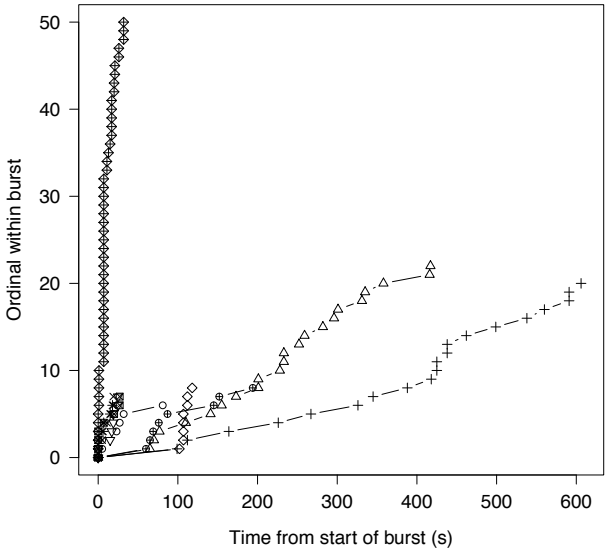


Figure 7: Development of failure bursts in one example cell.

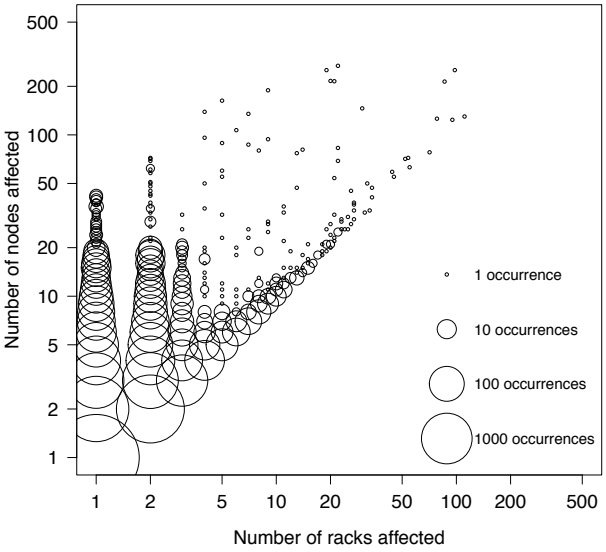


Figure 8: Frequency of failure bursts sorted by racks and nodes affected.

indicates the frequency of each burst group. The grouping of points along the 45° line represent bursts where as many racks are affected as nodes. The points furthest away from this line represent the most rack-correlated failure bursts. For larger bursts of at least 10 nodes, we find only 3% have all their nodes on unique racks. We introduce a metric to quantify this degree of domain correlation in the next section.

4.3 Identifying domain-related failures

Domain-related issues, such those associated with physical racks, network switches and power domains, are frequent causes of correlated failure. These problems can sometimes be difficult to detect directly. We introduce a metric to measure the likelihood that a failure burst is domain-related, rather than random, based on the pattern of failure observed. The metric can be used as an effective tool for identifying causes of failures that are connected to domain locality. It can also be used to evaluate the importance of domain diversity in cell design and data placement. We focus on detecting rack-related node failures in this section, but our methodology can be applied generally to any domain and any type of failure.

Let a failure burst be encoded as an n-tuple $(k_1, k_2, \ldots, k_n)$, where $k_1 \leq k_2 \leq \ldots \leq k_n$. Each k_i gives the number of nodes affected in the i-th rack affected, where racks are ordered so that these values are increasing. This *rack-based encoding* captures all relevant information about the rack locality of the burst. Let the *size* of the burst be the number of nodes that are affected, i.e., $\sum_{i=1}^{n} k_i$. We define the *rack-affinity score* of a burst to be

$$\sum_{i=1}^{n} \frac{k_i(k_i-1)}{2}$$

Note that this is the number of ways of choosing two nodes from the burst within the same rack. The score allows us to compare the rack concentration of bursts of the same size. For example the burst $(1, 4)$ has score 6. The burst $(1, 1, 1, 2)$ has score 1 which is lower. Therefore, the first burst is more concentrated by rack. Possible alternatives for the score include the sum of squares $\sum_{i=1}^{n} k_i^2$ or the negative entropy $\sum_{i=1}^{n} k_i \log(k_i)$. The sum of squares formula is equivalent to our chosen score because for a fixed burst size, the two formulas are related by an affine transform. We believe the entropy-inspired formula to be inferior because its log factor tends to downplay the effect of a very large k_i. Its real-valued score is also a problem for the dynamic program we use later in computation.

We define the *rack affinity* of a burst in a particular cell to be the probability that a burst of the same size affecting randomly chosen nodes in that cell will have a smaller burst score, plus half the probability that the two scores are equal, to eliminate bias. Rack affinity is therefore a number between 0 and 1 and can be interpreted as a vertical position on the cumulative distribution of the scores of random bursts of the same size. It can be shown that for a random burst, the expected value of its rack affinity is exactly 0.5. So we define a rack-correlated burst to be one with a metric close to 1, a rack-uncorrelated burst to be one with a metric close to 0.5, and a rack-anti-correlated burst to be one with a metric close to 0 (we have not observed such a burst). It is possible to ap-

proximate the metric using simulation of random bursts. We choose to compute the metric exactly using dynamic programming because the extra precision it provides allows us to distinguish metric values very close to 1.

We find that, in general, larger failure bursts have higher rack affinity. All our failure bursts of more than 20 nodes have rack affinity greater than 0.7, and those of more than 40 nodes have affinity at least 0.9. It is worth noting that some bursts with high rack affinity do not affect an entire rack and are not caused by common network or power issues. This could be the case for a bad batch of components or new storage node binary or kernel, whose installation is only slightly correlated with these domains.

5 Coping with Failure

We now begin the second part of the paper where we transition from node failures to analyzing replicated data availability. Two methods for coping with the large number of failures described in the first part of this paper include data replication and recovery, and chunk placement.

5.1 Data replication and recovery

Replication or erasure encoding schemes provide resilience to individual node failures. When a node failure causes the unavailability of a chunk within a stripe, we initiate a recovery operation for that chunk from the other available chunks remaining in the stripe.

Distributed filesystems will necessarily employ queues for recovery operations following node failure. These queues prioritize reconstruction of stripes which have lost the most chunks. The rate at which missing chunks may be recovered is limited by the bandwidth of individual disks, nodes, and racks. Furthermore, there is an explicit design tradeoff in the use of bandwidth for recovery operations versus serving client read/write requests.

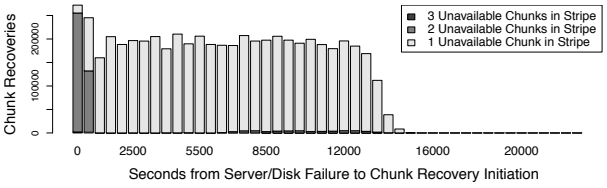


Figure 9: Example chunk recovery after failure bursts.

This limit is particularly apparent during correlated failures when a large number of chunks go missing at the same time. Figure 9 shows the recovery delay after a failure burst of 20 storage nodes affecting millions of stripes. Operators may adjust the rate-limiting seen in the figure.

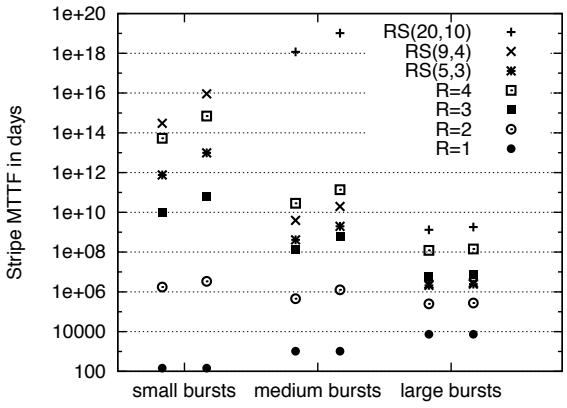


Figure 10: Stripe MTTF due to different burst sizes. Burst sizes are defined as a fraction of all nodes: small (0-0.001), medium (0.001-0.01), large (0.01-0.1). For each size, the left column represents uniform random placement, and the right column represents rack-aware placement.

The models presented in the following sections allow us to measure the sensitivity of data availability to this rate-limit and other parameters, described in Section 8.

5.2 Chunk placement and stripe unavailability

To mitigate the effect of large failure bursts in a single failure domain we consider known failure domains when placing chunks within a stripe on storage nodes. For example, racks constitute a significant failure domain to avoid. A rack-aware policy is one that ensures that no two chunks in a stripe are placed on nodes in the same rack.

Given a failure burst, we can compute the expected fraction of stripes made unavailable by the burst. More generally, we compute the probability that exactly k chunks are affected in a stripe of size n, which is essential to the Markov model of Section 7. Assuming that stripes are uniformly distributed across nodes of the cell, this probability is a ratio where the numerator is the number of ways to place a stripe of size n in the cell such that exactly k of its chunks are affected by the burst, and the denominator is the total number of ways to place a stripe of size n in the cell. These numbers can be computed combinatorially. The same ratio can be used when chunks are constrained by a placement policy, in which case the numerator and denominator are computed using dynamic programming.

Figure 10 shows the stripe MTTF for three classes of burst size. For each class of bursts we calculate the average fraction of stripes affected per burst and the rate of bursts, to get the combined MTTF due to that class. We see that for all encodings except $R = 1$, large failure bursts are the biggest contributor to unavailability

despite the fact that they are much rarer. We also see that for small and medium bursts sizes, and large encodings, using a rack-aware placement policy increases the stripe MTTF by a factor of 3 typically. This is a significant gain considering that in uniform random placement, most stripes end up with their chunks on different racks due to chance.

6 Cell Simulation

This section introduces a trace-based simulation method for calculating availability in a cell. The method replays observed or synthetic sequences of node failures and calculates the resulting impact on stripe availability. It offers detailed view of availability in short time frames.

For each node, the recorded events of interest are *down*, *up* and *recovery complete* events. When all nodes are up, they are each assumed to be responsible for an equal number of chunks. When a node goes down it is still responsible for the same number of chunks until 15 minutes later when the chunk recovery process starts. For simplicity and conservativeness, we assume that all these chunks remain unavailable until the *recovery complete* event. A more accurate model could model recovery too, such as by reducing the number of unavailable chunks linearly until the *recovery complete* event, or by explicitly modelling recovery queues.

We are interested in the expected number of stripes that are unavailable for at least 15 minutes, as a function of time. Instead of simulating a large number of stripes, it is more efficient to simulate all possible stripes, and use combinatorial calculations to obtain the expected number of unavailable stripes given a set of down nodes, as was done in Section 5.2.

As a validation, we can run the simulation using the stripe encodings that were in use at the time to see if the predicted number of unavailable stripes matches the actual number of unavailable stripes as measured by our storage system. Figure 11 shows the result of such a simulation. The prediction is a linear combination of the predictions for individual encodings present, in this case mostly $RS(5,3)$ and $R = 3$.

Analysis of hypothetical scenarios may also be made with the cell simulator, such as the effect of encoding choice and of chunk recovery rate. Although we may not change the frequency and severity of bursts in an observed sequence, bootstrap methods [13] may be used to generate synthetic failure traces with different burst characteristics. This is useful for exploring sensitivity to these events and the impact of improvements in datacenter reliability.

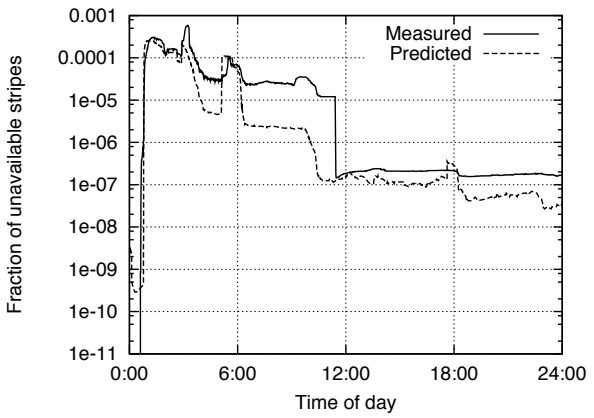


Figure 11: Unavailability prediction over time for a particular cell for a day with large failure bursts.

7 Markov Model of Stripe Availability

In this section, we formulate a Markov model of data availability. The model captures the interaction of different failure types and production parameters with more flexibility than is possible with the trace-based simulation described in the previous section. Although the model makes assumptions beyond those in the trace-based simulation method, it has certain advantages. First, it allows us to model and understand the impact of changes in hardware and software on end-user data availability. There are typically too many permutations of system changes and encodings to test each in a live cell. The Markov model allows us to reason directly about the contribution to data availability of each level of the storage stack and several system parameters, so that we can evaluate tradeoffs. Second, the systems we study may have unavailability rates that are so low they are difficult to measure directly. The Markov model handles rare events and arbitrarily low stripe unavailability rates efficiently.

The model focuses on the availability of a representative stripe. Let s be the total number of chunks in the stripe, and r be the minimum number of chunks needed to recover that stripe. As described in Section 2.2, $r = 1$ for replicated data and $r = n$ for $RS(n,m)$ encoded data. The state of a stripe is represented by the number of available chunks. Thus, the states are $s, s-1, \ldots, r, r-1$ with the state $r-1$ representing all of the *unavailable* states where the stripe has less than the required r chunks available. Figure 12 shows a Markov chain corresponding to an $R = 2$ stripe.

The Markov chain transitions are specified by the rates at which a stripe moves from one state to another, due to chunk failures and recoveries. Chunk failures reduce the number of available chunks, and several chunks may fail 'simultaneously' in a failure burst event. Balancing this, recoveries increase the number of available chunks if any

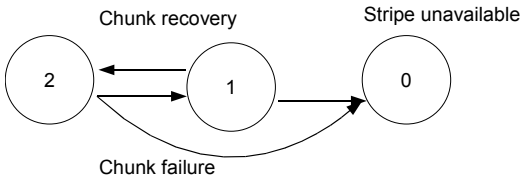


Figure 12: The Markov chain for a stripe encoded using $R = 2$.

are unavailable.

A key assumption of the Markov model is that events occur independently and with constant rates over time. This independence assumption, although strong, is not the same as the assumption that individual chunks fail independently of each other. Rather, it implies that failure events are independent of each other, but each event may involve multiple chunks. This allows a richer and more flexible view of the system. It also implies that recovery rates for a stripe depend only on its own current state.

In practice, failure events are not always independent. Most notably, it has been pointed out in [29] that the time between disk failures is not exponentially distributed and exhibits autocorrelation and long-range dependence. The Weibull distribution provides a much better fit for disk MTTF.

However, the exponential distribution is a reasonable approximation for the following reasons. First, the Weibull distribution is a generalization of the exponential distribution that allows the rate parameter to increase over time to reflect the aging of disks. In a large population of disks, the mixture of disks of different ages tends to be stable, and so the average failure rate in a cell tends to be constant. When the failure rate is stable, the Weibull distribution provides the same quality of fit as the exponential. Second, disk failures make up only a small subset of failures that we examined, and model results indicate that overall availability is not particularly sensitive to them. Finally, other authors ([24]) have concluded that correlation and non-homogeneity of the recovery rate and the mean time to a failure event have a much smaller impact on system-wide availability than the size of the event.

7.1 Construction of the Markov chain

We compute the transition rate due to failures using observed failure events. Let λ denote the rate of failure events affecting chunks, including node and disk failures. For any observed failure event we compute the probability that it affects k chunks out of the i available chunks in a stripe. As in Section 6, for failure bursts this computation takes into account the stripe placement strategy. The rate and severity of bursts, node, disk, and other failures may be adjusted here to suit the system parameters under exploration.

Averaging these probabilities over all failures events gives the probability, $p_{i,j}$, that a random failure event will affect $i-j$ out of i available chunks in a stripe. This gives a rate of transition from state i to state $j < i$, of $\lambda_{i,j} = \lambda p_{i,j}$ for $s \geq i > j \geq r$ and $\lambda_{i,r-1} = \lambda \sum_{j=0}^{r-1} p_{i,j}$ for the rate of reaching the unavailable state. Note that transitions from a state to itself are ignored.

For chunk recoveries, we assume a fixed rate of ρ for recovering a single chunk, i.e. moving from a state i to $i+1$, where $r \leq i < s$. In particular, this means we assume that the recovery rate does not depend on the total number of unavailable chunks in the cell. This is justified by setting ρ to a lower bound for the rate of recovery, based on observed recovery rates across our storage cells or proposed system performance parameters. While parallel recovery of multiple chunks from a stripe is possible, $\rho_{i,i+1} = (s-i)\rho$, we model serial recovery to gain more conservative estimates of stripe availability.

As with [12], the distributed systems we study use prioritized recovery for stripes with more than one chunk unavailable. Our Markov model allows state-dependent recovery that captures this prioritization, but for ease of exposition we do not use this added degree of freedom.

Finally, transition rates between pairs of states not mentioned are zero.

With the Markov chain thus completely specified, computing the MTTF of a stripe, as the mean time to reach the 'unavailable state' $r-1$ starting from state s, follows by standard methods [27].

7.2 Extension to multi-cell replication

The models introduced so far can be extended to compute the availability of multi-cell replication schemes. An example of such a scheme is $R = 3 \times 2$, where six replicas of the data are distributed as $R = 3$ replication in each of two linked cells. If data becomes unavailable at one cell then it is automatically recovered from another linked cell. These cells may be placed in separate datacenters, even on separate continents. Reed-Solomon codes may also be used, giving schemes such as $RS(6,3) \times 3$ for three cells each with a $RS(6,3)$ encoding of the data. We do not consider here the case when individual chunks may be combined from multiple cells to recover data, or other more complicated multi-cell encodings.

We compute the availability of stripes that span cells by building on the Markov model just presented. Intuitively, we treat each cell as a 'chunk' in the multi-cell 'stripe', and compute its availability using the Markov model. We assume that failures at different data centers are independent, that is, that they lack a single point of failure such as a shared power plant or network link. Ad-

ditionally, when computing the cell availability, we account for any cell-level or datacenter-level failures that would affect availability.

We build the corresponding transition matrix that models the resulting multi-cell availability as follows. We start from the transition matrices M_i for each cell, as explained in the previous section. We then build the transition matrix for the combined scheme as the tensor product of these, $\bigotimes_i M_i$, plus terms for whole cell failures, and for cross-cell recoveries if the data becomes unavailable in some cells but is still available in at least one cell. However, it is a fair approximation to simply treat each cell as a highly-reliable chunk in a multi-cell stripe, as described above.

Besides symmetrical cases, such as R = 3×2 replication, we can also model inhomogeneous replication schemes, such as one cell with $R = 3$ and one with $R = 2$. The state space of the Markov model is the product of the state space for each cell involved, but may be approximated again by simply counting how many of each type of cell is available.

A point of interest here is the recovery bandwidth between cells, quantified in Section 8.5. Bandwidth between distant cells has significant cost which should be considered when choosing a multi-cell replication scheme.

8 Markov Model Findings

In this section, we apply the Markov models described above to understand how changes in the parameters of the system will affect end-system availability.

8.1 Markov model validation

We validate the Markov model by comparing MTTF predicted by the model with actual MTTF values observed in production cells. We are interested in whether the Markov model provides an adequate tool for reasoning about stripe availability. Our main goal in using the model is providing a relative comparison of competing storage solutions, rather than a highly accurate prediction of any particular solution.

We underline two observations that surface from validation. First, the model is able to capture well the effect of failure bursts, which we consider as having the most impact on the availability numbers. For the cells we observed, the model predicted MTTF with the same order of magnitude as the measured MTTF. In one particular cell, besides more regular unavailability events, there was a large failure burst where tens of nodes became unavailable. This resulted in an MTTF of 1.76E+6 days, while the model predicted 5E+6 days. Though the relative error exceeds 100%, we are satisfied with the model accuracy, since it still gives us a powerful enough tool to make decisions, as can be seen in the following sections.

Second, the model can distinguish between failure bursts that span racks, and thus pose a threat to availability, and those that do not. If one rack goes down, then without other events in the cell, the availability of stripes with R=3 replication will not be affected, since the storage system ensures that chunks in each stripe are placed on different racks. For one example cell, we noticed tens of medium sized failure bursts that affected one or two racks. We expected the availability of the cell to stay high, and indeed we measured MTTF = 29.52E+8 days. The model predicted 5.77E+8 days. Again, the relative error is significant, but for our purposes the model provides sufficiently accurate predictions.

Validating the model for all possible replication and Reed-Solomon encodings is infeasible, since our production cells are not set up to cover the complete space of options. However, because of our large number of production cells we are able to validate the model over a range of encodings and operating conditions.

8.2 Importance of recovery rate

To develop some intuition about the sensitivity of stripe availability to recovery rate, consider the situation where there are no failure bursts. Chunks fail independently with rate λ and recover with rate ρ. As in the previous section, consider a stripe with s chunks total which can survive losing at most $s-r$ chunks, such as $RS(r, s-r)$. Thus the transition rate from state $i \geq r$ to state $i-1$ is $i\lambda$, and from state i to $i+1$ is ρ for $r \geq i < s$.

We compute the MTTF, given by the time taken to reach state $r-1$ starting in state s. Using standard methods related to *Gambler's Ruin*, [8, 14, 15, 26], this comes to:

$$\frac{1}{\lambda}\left(\sum_{k=0}^{s-r}\sum_{i=0}^{k}\frac{\rho^i}{\lambda^i}\frac{1}{(s-k+i)_{(i+1)}}\right)$$

where $(a)_{(b)}$ denotes $(a)(a-1)(a-2)\cdots(a-b+1)$.

Assuming recoveries take much less time than node MTTF (i.e. $\rho >> \lambda$), gives a stripe MTTF of:

$$\frac{\rho^{s-r}}{\lambda^{s-r+1}}\frac{1}{(s)_{(s-r+1)}} + O\left(\frac{\rho^{s-r-1}}{\lambda^{s-r}}\right)$$

By similar computations, the recovery bandwidth consumed is approximately λs per r data chunks.

Thus, with no correlated failures reducing recovery times by a factor of μ will increase stripe MTTF by a factor of μ^2 for $R = 3$ and by μ^4 for $RS(9, 4)$.

Reducing recovery times is effective when correlated failures are few. For $RS(6, 3)$ with no correlated failures, a 10% reduction in recovery time results in a 19% reduction in unavailability. However, when correlated failures

Policy (% overhead)	MTTF(days) with correlated failures	MTTF(days) w/o correlated failures
$R = 2$ (100)	$1.47E+5$	$4.99E+05$
$R = 3$ (200)	$6.82E+6$	$1.35E+09$
$R = 4$ (300)	$1.40E+8$	$2.75E+12$
$R = 5$ (400)	$2.41E+9$	$8.98E+15$
$RS(4,2)$ (50)	$1.80E+6$	$1.35E+09$
$RS(6,3)$ (50)	$1.03E+7$	$4.95E+12$
$RS(9,4)$ (44)	$2.39E+6$	$9.01E+15$
$RS(8,4)$ (50)	$5.11E+7$	$1.80E+16$

Table 3: Stripe MTTF in days, corresponding to various data redundancy policies and space overhead.

Policy (recovery time)	MTTF (days)	Bandwidth (per PB)
$R = 2 \times 2$(1day)	$1.08E+10$	6.8MB/day
$R = 2 \times 2$(1hr)	$2.58E+11$	6.8MB/day
$RS(6,3) \times 2$(1day)	$5.32E+13$	97KB/day
$RS(6,3) \times 2$(1hr)	$1.22E+15$	97KB/day

Table 4: Stripe MTTF and inter-cell bandwidth, for various multi-cell schemes and inter-cell recovery times.

are taken into account, even a 90% reduction in recovery time results in only a 6% reduction in unavailability.

8.3 Impact of correlation on effectiveness of data-replication schemes

Table 3 presents stripe availability for several data-replication schemes, measured in MTTF. We contrast this with stripe MTTF when node failures occur at the same total rate but are assumed independent.

Note that failing to account for correlation of node failures typically results in overestimating availability by at least two orders of magnitude, and eight in the case of RS(8,4). Correlation also reduces the benefit of increasing data redundancy. The gain in availability achieved by increasing the replication number, for example, grows much more slowly when we have correlated failures. Reed Solomon encodings achieve similar resilience to failures compared to replication, though with less storage overhead.

8.4 Sensitivity of availability to component failure rates

One common method for improving availability is reducing component failure rates. By inserting altered failure rates of hardware into the model we can estimate the impact of potential improvements without actually building or deploying new hardware.

We find that improvements below the node (server) layer of the storage stack do not significantly improve data availability. Assuming $R = 3$ is used, a 10% reduction in the latent disk error rate has a negligible effect on stripe availability. Similarly, a 10% reduction in the disk failure rate increases stripe availability by less than 1.5%. On the other hand, cutting node failure rates by 10% can increase data availability by 18%. This holds generally for other encodings.

8.5 Single vs multi-cell replication schemes

Table 4 compares stripe MTTF under several multi-cell replication schemes and inter-cell recovery times, taking into consideration the effect of correlated failures within cells.

Replicating data across multiple cells (data centers) greatly improves availability because it protects against correlated failures. For example, $R = 2 \times 2$ with 1 day recovery time between cells has two orders of magnitude longer MTTF than $R = 4$, shown in Table 3.

This introduces a tradeoff between higher replication in a single cell and the cost of inter-cell bandwidth. The extra availability for $R = 2 \times 2$ with 1 day recoveries versus $R = 4$ comes at an average cost of 6.8 MB/(user PB) copied between cells each day. This is the inverse MTTF for $R = 2$.

It should be noted that most cross-cell recoveries will occur in the event of large failure bursts. This must be considered when calculating expected recovery times between cells and the cost of on-demand access to potentially large amounts of bandwidth.

Considering the relative cost of storage versus recovery bandwidth allows us to choose the most cost effective scheme given particular availability goals.

9 Related Work

Several previous studies [3, 19, 25, 29, 30] focus on the failure characteristics of independent hardware components, such as hard drives, storage subsystems, or memory. As we have seen, these must be included when considering availability but by themselves are insufficient.

We focus on failure bursts, since they have a large influence on the availability of the system. Previous literature on failure bursts has focused on methods for discovering the relationship between the size of a failure event and its probability of occurrence. In [10], the existence of near-simultaneous failures in two large distributed systems is reported. The beta-binomial density and the bi-exponential density are used to fit these distributions in [6] and [24], respectively. In [24], the authors further note that using an over-simplistic model for burst size, for example a single size, could result in "dramatic inaccuracies" in practical settings. On the other hand, even

though the mean time to failure and mean time to recovery of system nodes tend to be non-uniform and correlated, this particular correlation effect has only a limited impact on system-wide availability.

There is limited previous work on discovering patterns of correlation in failures. The conditional probability of failures for each pair of nodes in a system has been proposed in [6] as a measure of correlation in the system. This computation extends heuristically to sets of larger nodes. A paradigm for discovering maximally independent groups of nodes in a system to cope with correlated failures is discussed in [34]. That paradigm involves collecting failure statistics on each node in the system and computing a measure of correlation, such as the mutual information, between every pair of nodes. Both of these approaches are computationally intensive and the results found, unlike ours, are not used to build a predictive analytical model for availability.

Models that have been developed to study the reliability of long-term storage fall into two categories, non-Markov and Markov models. Those in the first category tend to be less versatile. For example, in [5] the probability of multiple faults occurring during the recovery period of a stripe is approximated. Correlation is introduced by means of a multiplicative factor that is applied to the mean time to failure of a second chunk when the first chunk is already unavailable. This approach works only for stripes that are replicated and is not easily extendable to Reed-Solomon encoding. Moreover, the factor controlling time correlation is neither measurable nor derivable from other data.

In [33], replication is compared with Reed-Solomon with respect to storage requirement, bandwidth for write and repair and disk seeks for reads. However, the comparison assumes that sweep and repair are performed at regular intervals, as opposed to on demand.

Markov models are able to capture the system much more generally and can be used to model both replication and Reed-Solomon encoding. Examples include [21], [32], [11] and [35]. However, these models all assume independent failures of chunks. As we have shown, this assumption potentially leads to overestimation of data availability by many orders of magnitude. The authors of [20] build a tool to optimize the disaster recovery according to availability requirements, with similar goals as our analysis of multi-cell replication. However, they do not focus on studying the effect of failure characteristics and data redundancy options.

Node availability in our environment is different from previous work, such as [7, 18, 23], because we study a large system that is tightly coupled in a single administrative domain. These studies focus on measuring and predicting availability of individual desktop machines from many, potentially untrusted, domains. Other authors [11] studied data replication in face of failures, though without considering availability of Reed-Solomon encodings or multi-cell replication.

10 Conclusions

We have presented data from Google's clusters that characterize the sources of failures contributing to unavailability. We find that correlation among node failures dwarfs all other contributions to unavailability in our production environment.

In particular, though disks failures can result in permanent data loss, the multitude of transitory node failures account for most unavailability. We present a simple time-window-based method to group failure events into failure bursts which, despite its simplicity, successfully identifies bursts with a common cause. We develop analytical models to reason about past and future availability in our cells, including the effects of different choices of replication, data placement and system parameters.

Inside Google, the analysis described in this paper has provided a picture of data availability at a finer granularity than previously measured. Using this framework, we provide feedback and recommendations to the development and operational engineering teams on different replication and encoding schemes, and the primary causes of data unavailability in our existing cells. Specific examples include:

- Determining the acceptable rate of successful transfers to battery power for individual machines upon a power outage.
- Focusing on reducing reboot times, because planned kernel upgrades are a major source of correlated failures.
- Moving towards a dynamic delay before initiating recoveries, based on failure classification and recent history of failures in the cell.

Such analysis complements the intuition of the designers and operators of these complex distributed systems.

Acknowledgments

Our findings would not have been possible without the help of many of our colleagues. We would like to thank the following people for their contributions to data collection: Marc Berhault, Eric Dorland, Sangeetha Eyunni, Adam Gee, Lawrence Greenfield, Ben Kochie, and James O'Kane. We would also like to thank a number of our colleagues for helping us improve the presentation of these results. In particular, feedback from John Wilkes, Tal Garfinkel, and Mike Marty was helpful. We

would also like to thank our shepherd Bianca Schroeder and the anonymous reviewers for their excellent feedback and comments, all of which helped to greatly improve this paper.

References

[1] HDFS (Hadoop Distributed File System) architecture. `http://hadoop.apache.org/common/docs/current/hdfs_design.html`, 2009.

[2] ANDREAS, E. S., HAEBERLEN, A., DABEK, F., GON CHUN, B., WEATHERSPOON, H., MORRIS, R., KAASHOEK, M. F., AND KUBIATOWICZ, J. Proactive replication for data durability. In *Proceedings of the 5th Intl Workshop on Peer-to-Peer Systems (IPTPS)* (2006).

[3] BAIRAVASUNDARAM, L. N., GOODSON, G. R., PASUPATHY, S., AND SCHINDLER, J. An analysis of latent sector errors in disk drives. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2007), pp. 289–300.

[4] BAIRAVASUNDARAM, L. N., GOODSON, G. R., SCHROEDER, B., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEA, R. H. An analysis of data corruption in the storage stack. In *FAST '08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), pp. 1–16.

[5] BAKER, M., SHAH, M., ROSENTHAL, D. S. H., ROUSSOPOULOS, M., MANIATIS, P., GIULI, T., AND BUNGALE, P. A fresh look at the reliability of long-term digital storage. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems* (2006), pp. 221–234.

[6] BAKKALOGLU, M., WYLIE, J. J., WANG, C., AND GANGER, G. R. Modeling correlated failures in survivable storage systems. In *Fast Abstract at International Conference on Dependable Systems & Networks* (June 2002).

[7] BOLOSKY, W. J., DOUCEUR, J. R., ELY, D., AND THEIMER, M. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on measurement and modeling of computer systems* (2000), pp. 34–43.

[8] BROWN, D. M. The first passage time distribution for a parallel exponential system with repair. In *Reliability and fault tree analysis* (1974), Defense Technical Information Center.

[9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Nov. 2006), pp. 205–218.

[10] CHARACTERISTICS, M. F., YALAG, P., NATH, S., YU, H., GIBBONS, P. B., AND SESHAN, S. Beyond availability: Towards a deeper understanding of machine failure characteristics in large distributed systems. In *WORLDS '04: First Workshop on Real, Large Distributed Systems* (2004).

[11] CHUN, B.-G., DABEK, F., HAEBERLEN, A., SIT, E., WEATHERSPOON, H., KAASHOEK, M. F., KUBIATOWICZ, J., AND MORRIS, R. Efficient replica maintenance for distributed storage systems. In *NSDI '06: Proceedings of the 3rd conference on Networked Systems Design & Implementation* (2006), pp. 45–58.

[12] CORBETT, P., ENGLISH, B., GOEL, A., GRCANAC, T., KLEIMAN, S., LEONG, J., AND SANKAR, S. Row-diagonal parity for double disk failure correction. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004), pp. 1–14.

[13] EFRON, B., AND TIBSHIRANI, R. *An Introduction to the Bootstrap*. Chapman and Hall, 1993.

[14] EPSTEIN, R. *The Theory of Gambling and Statistical Logic*. Academic Press, 1977.

[15] FELLER, W. *An Introduction to Probability Theory and Its Application*. John Wiley and Sons, 1968.

[16] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Oct. 2003), pp. 29–43.

[17] ISARD, M. Autopilot: automatic data center management. *ACM SIGOPS Opererating Systems Review 41*, 2 (2007), 60–67.

[18] JAVADI, B., KONDO, D., VINCENT, J.-M., AND ANDERSON, D. Mining for statistical models of availability in large-scale distributed systems: An empirical study of SETI@home (2009). pp. 1–10.

[19] JIANG, W., HU, C., ZHOU, Y., AND KANEVSKY, A. Are disks the dominant contributor for storage failures?: a comprehensive study of storage subsystem failure characteristics. In *FAST '08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), pp. 1–15.

[20] KEETON, K., SANTOS, C., BEYER, D., CHASE, J., AND WILKES, J. Designing for disasters. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004), pp. 59–62.

[21] LIAN, Q., CHEN, W., AND ZHANG, Z. On the impact of replica placement to the reliability of distributed brick storage systems. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems* (2005), pp. 187–196.

[22] MCKUSICK, M. K., AND QUINLAN, S. GFS: Evolution on fast-forward. *Communications of the ACM 53*, 3 (2010), 42–49.

[23] MICKENS, J. W., AND NOBLE, B. D. Exploiting availability prediction in distributed systems. In *NSDI '06: Proceedings of the 3rd conference on Networked Systems Design & Implementation* (2006), pp. 73–86.

[24] NATH, S., YU, H., GIBBONS, P. B., AND SESHAN, S. Subtleties in tolerating correlated failures in wide-area storage systems. In *NSDI '06: Proceedings of the 3rd conference on Networked Systems Design & Implementation* (2006), pp. 225–238.

[25] PINHEIRO, E., WEBER, W.-D., AND BARROSO, L. A. Failure trends in a large disk drive population. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies* (2007), pp. 17–23.

[26] RAMABHADRAN, S., AND PASQUALE, J. Analysis of long-running replicated systems. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications* (2006), pp. 1–9.

[27] RESNICK, S. I. *Adventures in stochastic processes*. Birkhauser Verlag, 1992.

[28] RODRIGUES, R., AND LISKOV, B. High availability in DHTs: Erasure coding vs. replication. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems* (2005).

[29] SCHROEDER, B., AND GIBSON, G. A. Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you? In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies* (2007), pp. 1–16.

[30] SCHROEDER, B., PINHEIRO, E., AND WEBER, W.-D. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS '09: Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems* (2009), pp. 193–204.

[31] SCHWARZ, T. J. E., XIN, Q., MILLER, E. L., LONG, D. D. E., HOSPODOR, A., AND NG, S. Disk scrubbing in large archival storage systems. *International Symposium on Modeling, Analysis, and Simulation of Computer Systems* (2004), 409–418.

[32] T., S. Generalized Reed Solomon codes for erasure correction in SDDS. In *WDAS-4: Workshop on Distributed Data and Structures* (2002).

[33] WEATHERSPOON, H., AND KUBIATOWICZ, J. Erasure coding vs. replication: A quantitative comparison. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems* (2002), Springer-Verlag, pp. 328–338.

[34] WEATHERSPOON, H., MOSCOVITZ, T., AND KUBIATOWICZ, J. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems* (2002), pp. 362–367.

[35] XIN, Q., MILLER, E. L., SCHWARZ, T., LONG, D. D. E., BRANDT, S. A., AND LITWIN, W. Reliability mechanisms for very large storage systems. In *MSS '03: Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies* (2003), pp. 146–156.

Nectar: Automatic Management of Data and Computation in Datacenters

Pradeep Kumar Gunda, Lenin Ravindranath*, Chandramohan A. Thekkath, Yuan Yu, Li Zhuang

Microsoft Research Silicon Valley

Abstract

Managing data and computation is at the heart of datacenter computing. Manual management of data can lead to data loss, wasteful consumption of storage, and laborious bookkeeping. Lack of proper management of computation can result in lost opportunities to share common computations across multiple jobs or to compute results incrementally.

Nectar is a system designed to address the aforementioned problems. It automates and unifies the management of data and computation within a datacenter. In Nectar, data and computation are treated interchangeably by associating data with its computation. Derived datasets, which are the results of computations, are uniquely identified by the programs that produce them, and together with their programs, are automatically managed by a datacenter wide caching service. Any derived dataset can be transparently regenerated by re-executing its program, and any computation can be transparently avoided by using previously cached results. This enables us to greatly improve datacenter management and resource utilization: obsolete or infrequently used derived datasets are automatically garbage collected, and shared common computations are computed only once and reused by others.

This paper describes the design and implementation of Nectar, and reports on our evaluation of the system using analytic studies of logs from several production clusters and an actual deployment on a 240-node cluster.

1 Introduction

Recent advances in distributed execution engines (MapReduce [7], Dryad [18], and Hadoop [12]) and high-level language support (Sawzall [25], Pig [24], BOOM [3], HIVE [17], SCOPE [6], DryadLINQ [29]) have greatly simplified the development of large-scale, data-intensive, distributed applications. However, major challenges still remain in realizing the full potential of data-intensive distributed computing within datacenters. In current practice, a large fraction of the computations in a datacenter is redundant and many datasets are obsolete or seldom used, wasting vast amounts of resources in a datacenter.

As one example, we quantified the wasted storage in our 240-node experimental Dryad/DryadLINQ cluster. We crawled this cluster and noted the last access time for each data file. We discovered that around 50% of the files was not accessed in the last 250 days.

As another example, we examined the execution statistics of 25 production clusters running data-parallel applications. We estimated that, on one such cluster, over 7000 hours of redundant computation can be eliminated per day by caching intermediate results. (This is approximately equivalent to shutting off 300 machines daily.) Cumulatively, over all clusters, this figure is over 35,000 hours per day.

Many of the resource issues in a datacenter arise due to lack of efficient management of either data or computation, or both. This paper describes Nectar: a system that manages the execution environment of a datacenter and is designed to address these problems.

A key feature of Nectar is that it treats data and computation in a datacenter interchangeably in the following sense. Data that has not been accessed for a long period may be removed from the datacenter and substituted by the computation that produced it. Should the data be needed in the future, the computation is rerun. Similarly, instead of executing a user's program, Nectar can partially or fully substitute the results of that computation with data already present in the datacenter. Nectar relies on certain properties of the programming environment in the datacenter to enable this interchange of data and computation.

Computations running on a Nectar-managed datacen-

*L. Ravindranath is affiliated with the Massachusetts Institute of Technology and was a summer intern on the Nectar project.

ter are specified as programs in LINQ [20]. LINQ comprises a set of operators to manipulate datasets of .NET objects. These operators are integrated into high level .NET programming languages (e.g., C#), giving programmers direct access to .NET libraries as well traditional language constructs such as loops, classes, and modules. The datasets manipulated by LINQ can contain objects of an arbitrary .NET type, making it easy to compute with complex data such as vectors, matrices, and images. All of these operators are *functional*: they transform input datasets to new output datasets. This property helps Nectar reason about programs to detect program and data dependencies. LINQ is a very expressive and flexible language, e.g., the MapReduce class of computations can be trivially expressed in LINQ.

Data stored in a Nectar-managed datacenter are divided into one of two classes: *primary* or *derived*. Primary datasets are created once and accessed many times. Derived datasets are the results produced by computations running on primary and other derived datasets. Examples of typical primary datasets in our datacenters are click and query logs. Examples of typical derived datasets are the results of thousands of computations performed on those click and query logs.

In a Nectar-managed datacenter, all access to a derived dataset is mediated by Nectar. At the lowest level of the system, a derived dataset is referenced by the LINQ program fragment or expression that produced it. Programmers refer to derived datasets with simple pathnames that contain a simple indirection (much like a UNIX symbolic link) to the actual LINQ programs that produce them. By maintaining this mapping between a derived dataset and the program that produced it, Nectar can reproduce any derived dataset after it is automatically deleted. Primary datasets are referenced by conventional pathnames, and are not automatically deleted.

A Nectar-managed datacenter offers the following advantages.

1. Efficient space utilization. Nectar implements a cache server that manages the storage, retrieval, and eviction of the results of all computations (i.e., derived datasets). As well, Nectar retains the description of the computation that produced a derived dataset. Since programmers do not directly manage datasets, Nectar has considerable latitude in optimizing space: it can remove unused or infrequently used derived datasets and recreate them on demand by rerunning the computation. This is a classic trade-off of storage and computation.

2. Reuse of shared sub-computations. Many applications running in the same datacenter share common sub-computations. Since Nectar automatically caches the results of sub-computations, they will be computed only once and reused by others. This significantly reduces redundant computations, resulting in better resource utilization.

3. Incremental computations. Many datacenter applications repeat the same computation on a sliding window of an incrementally augmented dataset. Again, caching in Nectar enables us to reuse the results of old data and only compute incrementally for the newly arriving data.

4. Ease of content management. With derived datasets uniquely named by LINQ expressions, and automatically managed by Nectar, there is little need for developers to manage their data manually. In particular, they do not have to be concerned about remembering the location of the data. Executing the LINQ expression that produced the data is sufficient to access the data, and incurs negligible overhead in almost all cases because of caching. This is a significant advantage because most datacenter applications consume a large amount of data from diverse locations and keeping track of the requisite filepath information is often a source of bugs.

Our experiments show that Nectar, on average, could improve space utilization by at least 50%. As well, incremental and sub-computations managed by Nectar provide an average speed up of 30% for the programs running on our clusters. We provide a detailed quantitative evaluation of the first three benefits in Section 4. We have not done a detailed user study to quantify the fourth benefit, but the experience from our initial deployment suggests that there is evidence to support the claim.

Some of the techniques we used such as dividing datasets into primary and derived and reusing the results of previous computations via caching are reminiscent of earlier work in version management systems [15], incremental database maintenance [5], and functional caching [16, 27]. Section 5 provides a more detailed analysis of our work in relation to prior research.

This paper makes the following contributions to the literature:

- We propose a novel and promising approach that automates and unifies the management of data and computation in a datacenter, leading to substantial improvements in datacenter resource utilization.

- We present the design and implementation of our system, including a sophisticated program rewriter and static program dependency analyzer.

- We present a systematic analysis of the performance of our system from a real deployment on 240-nodes as well as analytical measurements.

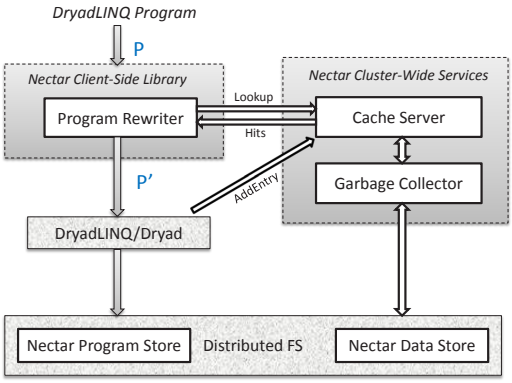


Figure 1: Nectar architecture. The system consists of a client-side library and cluster-wide services. Nectar relies on the services of DryadLINQ/Dryad and TidyFS, a distributed file system.

The rest of this paper is organized as follows. Section 2 provides a high-level overview of the Nectar system. Section 3 describes the implementation of the system. Section 4 evaluates the system using real workloads. Section 5 covers related work and Section 6 discusses future work and concludes the paper.

2 System Design Overview

The overall Nectar architecture is shown in Figure 1. Nectar consists of a client-side component that runs on the programmer's desktop, and two services running in the datacenter.

Nectar is completely transparent to user programs and works as follows. It takes a DryadLINQ program as input, and consults the cache service to rewrite it to an equivalent, more efficient program. Nectar then hands the resulting program to DryadLINQ which further compiles it into a Dryad computation running in the cluster. At run time, a Dryad job is a directed acyclic graph where vertices are programs and edges represent data channels. Vertices communicate with each other through data channels. The input and output of a DryadLINQ program are expected to be *streams*. A stream consists of an ordered sequence of extents, each storing a sequence of object of some data type. We use an in-house fault-tolerant, distributed file system called TidyFS to store streams.

Nectar makes certain assumptions about the underlying storage system. We require that streams be append-only, meaning that new contents are added by either appending to the last extent or adding a new extent. The metadata of a stream contains Rabin fingerprints [4] of the entire stream and its extents.

Nectar maintains and manages two namespaces in TidyFS. The program store keeps all DryadLINQ programs that have ever executed successfully. The data store is used to store all derived streams generated by DryadLINQ programs. The Nectar cache server provides cache hits to the program rewriter on the client side. It also implements a replacement policy that deletes cache entries of least value. Any stream in the data store that is not referenced by any cache entry is deemed to be garbage and deleted permanently by the Nectar garbage collector. Programs in the program store are never deleted and are used to recreate a deleted derived stream if it is needed in the future.

A simple example of a program is shown in Example 2.1. The program groups identical words in a large document into groups and applies an arbitrary user-defined function `Reduce` to each group. This is a typical MapReduce program. We will use it as a running example to describe the workings of Nectar. TidyFS, Dryad, and DryadLINQ are described in detail elsewhere [8, 18, 29]. We only discuss them briefly below to illustrate their relationships to our system.

In the example, we assume that the input D is a large (replicated) dataset partitioned as D_1, D_2 ... D_n in the TidyFS distributed file system and it consists of lines of text. `SelectMany` is a LINQ operator, which first produces a single list of output records for each input record and then "flattens" the lists of output records into a single list. In our example, the program applies the function `x => x.Split(' ')` to each line in D to produce the list of words in D.

The program then uses the `GroupBy` operator to group the words into a list of groups, putting the same words into a single group. `GroupBy` takes a *key-selector* function as the argument, which when applied to an input record returns a collating "key" for that record. `GroupBy` applies the key-selector function to each input record and collates the input into a list of groups (multi-sets), one group for all the records with the same key.

The last line of the program applies a transformation `Reduce` to each group. `Select` is a simpler version of `SelectMany`. Unlike the latter, `Select` produces a single output record (determined by the function `Reduce`) for each input record.

Example 2.1 A typical MapReduce job expressed in LINQ. `(x => x.Split(' '))` produces a list of blank-separated words; `(x => x)` produces a key for each input; `Reduce` is an arbitrary user supplied function that is applied to each input.

```
words = D.SelectMany(x => x.Split(' '));
groups = words.GroupBy(x => x);
result = groups.Select(x => Reduce(x));
```

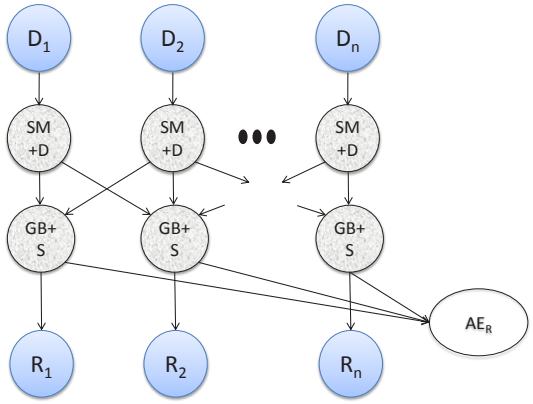


Figure 2: Execution graph produced by Nectar given the input LINQ program in Example 2.1. The nodes named `SM+D` executes `SelectMany` and distributes the results. `GB+S` executes `GroupBy` and `Select`.

When the program in Example 2.1 is run for the first time, Nectar, by invoking DryadLINQ, produces the distributed execution graph shown in Figure 2, which is then handed to Dryad for execution. (For simplicity of exposition, we assume for now that there are no cache hits when Nectar rewrites the program.) The `SM+D` vertex performs the `SelectMany` and distributes the results by partitioning them on a hash of each word. This ensures that identical words are destined to the same `GB+S` vertex in the graph. The `GB+S` vertex performs the `GroupBy` and `Select` operations together. The `AE` vertex adds a cache entry for the final result of the program. Notice that the derived stream created for the cache entry shares the same set of extents with the result of the computation. So, there is no additional cost of storage space. As a rule, Nectar always creates a cache entry for the final result of a computation.

2.1 Client-Side Library

On the client side, Nectar takes advantage of cached results from the cache to rewrite a program P to an equivalent, more efficient program P'. It automatically inserts *AddEntry* calls at appropriate places in the program so new cache entries can be created when P' is executed. The AddEntry calls are compiled into Dryad vertices that create new cache entries at runtime. We summarize the two main client-side components below.

Cache Key Calculation

A computation is uniquely identified by its program and inputs. We therefore use the Rabin fingerprint of the program and the input datasets as the cache key for a computation. The input datasets are stored in TidyFS and their fingerprints are calculated based on the actual stream contents. Nectar calculates the fingerprint of the program and combines it with the fingerprints of the input datasets to form the cache key.

The fingerprint of a DryadLINQ program must be able to detect any changes to the code the program depends on. However, the fingerprint should not change when code the program does not depend on changes. This is crucial for the correctness and practicality of Nectar. (Fingerprints can collide but the probability of a collision can be made vanishingly small by choosing long enough fingerprints.) We implement a static dependency analyzer to compute the transitive closure of all the code that can be reached from the program. The fingerprint is then formed using all reachable code. Of course, our analyzer only produces an over-approximation of the true dependency.

Rewriter

Nectar rewrites user programs to use cached results where possible. We might encounter different entries in the cache server with different sub-expressions and/or partial input datasets. So there are typically multiple alternatives to choose from in rewriting a DryadLINQ program. The rewriter uses a cost estimator to choose the best one from multiple alternatives (as discussed in Section 3.1).

Nectar supports the following two rewriting scenarios that arise very commonly in practice.

Common sub-expressions. Internally, a DryadLINQ program is represented as a LINQ expression tree. Nectar treats all prefix sub-expressions of the expression tree as candidates for caching and looks up in the cache for possible cache hits for every prefix sub-expression.

Incremental computations. Incremental computation on datasets is a common occurrence in data intensive computing. Typically, a user has run a program P on input D. Now, he is about to compute P on input $D + D'$, the concatenation of D and D'. The Nectar rewriter finds a new operator to combine the results of computing on the old input and the new input separately. See Section 2.3 for an example.

A special case of incremental computation that occurs in datacenters is a computation that executes on a sliding window of data. That is, the same program is repeatedly run on the following sequence of inputs:

$$Input_1 = d_1 + d_2 + ... + d_n,$$
$$Input_2 = d_2 + d_3 + ... + d_{n+1},$$
$$Input_3 = d_3 + d_4 + ... + d_{n+2},$$
$$.....$$

Here d_i is a dataset that (potentially) consists of multiple extents distributed over many computers. So successive inputs to the program ($Input_i$) are datasets with some old extents removed from the head of the previous input and new extents appended to the tail of it. Nectar generates cache entries for each individual dataset d_i, and can use them in subsequent computations.

In the real world, a program may belong to a combination of the categories above. For example, an application that analyzes logs of the past seven days is rewritten as an incremental computation by Nectar, but Nectar may use sub-expression results of log preprocessing on each day from other applications.

2.2 Datacenter-Wide Service

The datacenter-wide service in Nectar comprises two separate components: the cache service and the garbage collection service. The actual datasets are stored in the distributed storage system and the datacenter-wide services manipulate the actual datasets by maintaining pointers to them.

Cache Service

Nectar implements a distributed datacenter-wide cache service for bookkeeping information about DryadLINQ programs and the location of their results. The cache service has two main functionalities: (1) serving the cache lookup requests by the Nectar rewriter; and (2) managing derived datasets by deleting the cache entries of least value.

Programs of all successful computations are uploaded to a dedicated program store in the cluster. Thus, the service has the necessary information about cached results, meaning that it has a recipe to recreate any derived dataset in the datacenter. When a derived dataset is deleted but needed in the future, Nectar recreates it using the program that produced it. If the inputs to that program have themselves been deleted, it backtracks recursively till it hits the immutable primary datasets or cached derived datasets. Because of this ability to recreate datasets, the cache server can make informed decisions to implement a cache replacement policy, keeping the cached results that yield the most hits and deleting the cached results of less value when storage space is low.

Garbage Collector

The Nectar garbage collector operates transparently to the users of the cluster. Its main job is to identify datasets unreachable from any cache entry and delete them. We use a standard mark-and-sweep collector. Actual content deletion is done in the background without interfering with the concurrent activities of the cache server and job executions. Section 3.2 has additional detail.

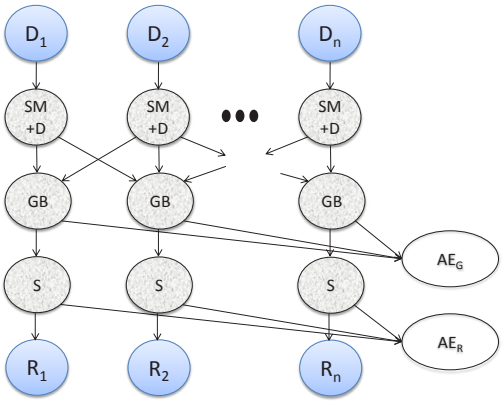


Figure 3: Execution graph produced by Nectar on the program in Example 2.1 after it elects to cache the results of computations. Notice that the `GroupBy` and `Select` are now encapsulated in separate nodes. The new `AE` vertex creates a cache entry for the output of `GroupBy`.

2.3 Example: Program Rewriting

Let us look at the interesting case of incremental computation by continuing Example 2.1.

After the program has been executed a sufficient number of times, Nectar may elect to cache results from some of its subcomputations based on the usage information returned to it from the cache service. So subsequent runs of the program may cause Nectar to create different execution graphs than those created previously for the same program. Figure 3 shows the new execution graph when Nectar chooses to cache the result of `GroupBy` (c.f. Figure 2). It breaks the pipeline of `GroupBy` and `Select` and creates an additional AddEntry vertex (denoted by `AE`) to cache the result of `GroupBy`. During the execution, when the `GB` stage completes, the `AE` vertex will run, creating a new TidyFS stream and a cache entry for the result of `GroupBy`. We denote the stream by G_D, partitioned as G_{D_1}, G_{D_2}, .. G_{D_n}.

Subsequently, assume the program in Example 2.1 is run on input $(D + X)$, where X is a new dataset partitioned as X_1, X_2,.. X_k. The Nectar rewriter would get a cache hit on G_D. So it only needs to perform `GroupBy` on X and merge with G_D to form new groups. Figure 4 shows the new execution graph created by Nectar.

There are some subtleties involved in the rewriting process. Nectar first determines that the number of partitions (n) of G_D. It then computes `GroupBy` on X the same way as G_D, generating n partitions with the same distribution scheme using the identical hash function as was used previously (see Figures 2 and 3). That is, the rewritten execution graph has k `SM+D` vertices, but n `GB`

vertices. The MG vertex then performs a pairwise merge of the output GB with the cached result G_D. The result of MG is again cached for future uses, because Nectar notices the pattern of incremental computation and expects that the same computation will happen on datasets of form G_{D+X+Y} in the future.

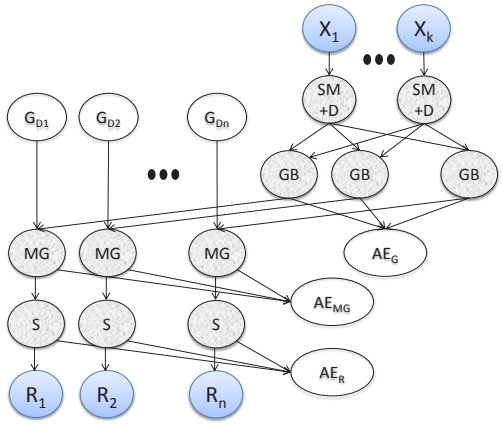


Figure 4: The execution graph produced by Nectar on the program in Example 2.1 on the dataset $D + X$. The dataset X consists of k partitions. The MG vertex merges groups with the same key. Both the results of GB and MG are cached. There are k SM+D vertices, but n GB, MG, and S vertices. $G_{D1}, ..., G_{Dn}$ are the partitions of the cached result.

Similar to MapReduce's combiner optimization [7] and Data Cube computation [10], DryadLINQ can decompose Reduce into the composition of two associative and commutative functions if Reduce is determined to be decomposable. We handle this by first applying the decomposition as in [28] and then the caching and rewriting as described above.

3 Implementation Details

We now present the implementation details of the two most important aspects of Nectar: Section 3.1 describes computation caching and Section 3.2 describes the automatic management of derived datasets.

3.1 Caching Computations

Nectar rewrites a DryadLINQ program to an equivalent but more efficient one using cached results. This generally involves: 1) identifying all sub-expressions of the expression, 2) probing the cache server for all cache hits for the sub-expressions, 3) using the cache hits to rewrite the expression into a set of equivalent expressions, and 4) choosing one that gives us the maximum benefit based on some cost estimation.

Cache and Programs

A cache entry records the result of executing a program on some given input. (Recall that a program may have more than one input depending on its arity.) The entry is of the form:

$$\langle FP_{PD}, FP_P, Result, Statistics, FPList \rangle$$

Here, FP_{PD} is the combined fingerprint of the program and its input datasets, FP_P is the fingerprint of the program only, $Result$ is the location of the output, and $Statistics$ contains execution and usage information of this cache entry. The last field $FPList$ contains a list of fingerprint pairs each representing the fingerprints of the first and last extents of an input dataset. We have one fingerprint pair for every input of the program. As we shall see later, it is used by the rewriter to search amongst cache hits efficiently. Since the same program could have been executed on different occasions on different inputs, there can be multiple cache entries with the same FP_P.

We use FP_{PD} as the primary key. So our caching is sound only if FP_{PD} can uniquely determine the result of the computation. The fingerprint of the inputs is based on the actual content of the datasets. The fingerprint of a dataset is formed by combining the fingerprints of its extents. For a large dataset, the fingerprints of its extents are efficiently computed in parallel by the datacenter computers.

The computation of the program fingerprint is tricky, as the program may contain user-defined functions that call into library code. We implemented a static dependency analyzer to capture all dependencies of an expression. At the time a DryadLINQ program is invoked, DryadLINQ knows all the dynamic linked libraries (DLLs) it depends on. We divide them into two categories: system and application. We assume system DLLs are available and identical on all cluster machines and therefore are not included in the dependency. For an application DLL that is written in native code (e.g., C or assembler), we include the entire DLL as a dependency. For soundness, we assume that there are no callbacks from native to managed code. For an application DLL that is in managed code (e.g., C#), our analyzer traverses the call graph to compute all the code reachable from the initial expression.

The analyzer works at the bytecode level. It uses standard .NET reflection to get the body of a method, finds all the possible methods that can be called in the body, and traverses those methods recursively. When a virtual method call is encountered, we include all the possible call sites. While our analysis is certainly a conservative approximation of the true dependency, it is reasonably

precise and works well in practice. Since dynamic code generation could introduce unsoundness into the analysis, it is forbidden in managed application DLLs, and is statically enforced by the analyzer.

The statistics information kept in the cache entry is used by the rewriter to find an *optimal* execution plan. It is also used to implement the cache insertion and eviction policy. It contains information such as the *cumulative execution time*, the number of hits on this entry, and the last access time. The cumulative execution time is defined as the sum of the execution time of all upstream Dryad vertices of the current execution stage. It is computed at the time of the cache entry insertion using the execution logs generated by Dryad.

The cache server supports a simple client interface. The important operations include: (1) `Lookup(fp)` finds and returns the cache entry that has `fp` as the primary key (FP_{PD}); (2) `Inquire(fp)` returns all cache entries that have `fp` as their FP_P; and (3) `AddEntry` inserts a new cache entry. We will see their uses in the following sections.

The Rewriting Algorithm

Having explained the structure and interface of the cache, let us now look at how Nectar rewrites a program.

For a given expression, we may get cache hits on any possible sub-expression and subset of the input dataset, and considering all of them in the rewriting is not tractable. We therefore only consider cache hits on prefix sub-expressions on segments of the input dataset. More concretely, consider a simple example `D.Where(P).Select(F)`. The `Where` operator applies a filter to the input dataset `D`, and the `Select` operator applies a transformation to each item in its input. We will only consider cache hits for the sub-expressions `S.Where(P)` and `S.Where(P).Select(F)` where `S` is a subsequence of extents in `D`.

Our rewriting algorithm is a simple recursive procedure. We start from the largest prefix sub-expression, the entire expression. Below is an outline of the algorithm. For simplicity of exposition, we assume that the expressions have only one input.

Step 1. For the current sub-expression E, we probe the cache server to obtain all the possible hits on it. There can be multiple hits on different subsequences of the input `D`. Let us denote the set of hits by H. Note that each hit also gives us its saving in terms of cumulative execution time. If there is a hit on the entire input `D`, we use that hit and terminate because it gives us the most savings in terms of cumulative execution time. Otherwise we execute Steps 2-4.

Step 2. We compute the best execution plan for E using hits on its smaller prefixes. To do that, we first compute the best execution plan for each immediate successor prefix of E by calling our procedure recursively, and then combine them to form a single plan for E. Let us denote this plan by (P_1, C_1) where C_1 is its saving in terms of cumulative execution time.

Step 3. For the H hits on E (from Step 1), we choose a subset of them such that (a) they operate on disjoint subsequence of `D`, and (b) they give us the most saving in terms of cumulative execution time. This boils down to the well-known problem of computing the maximum independent sets of an interval graph, which has a known efficient solution using dynamic programming techniques [9]. We use this subset to form another execution plan for E on `D`. Let us denote this plan by (P_2, C_2).

Step 4. The final execution plan is the one from P_1 and P_2 that gives us more saving.

In Step 1, the rewriter calls `Inquire` to compute H. As described before, `Inquire` returns all the possible cache hits of the program with different inputs. A useful hit means that its input dataset is identical to a subsequence of extents of `D`. A brute force search is inefficient and requires to check every subsequence. As an optimization, we store in the cache entry the fingerprints of the first and last extents of the input dataset. With that information, we can compute H in linear time.

Intuitively, in rewriting a program P on incremental data Nectar tries to derive a combining operator C such that $P(D+D') = C(P(D), D')$, where C combines the results of P on the datasets D and D'. Nectar supports all the LINQ operators DryadLINQ supports.

The combining functions for some LINQ operators require the parallel merging of multiple streams, and are not directly supported by DryadLINQ. We introduced three combining functions: `MergeSort`, `HashMergeGroups`, and `SortMergeGroups`, which are straightforward to implement using DryadLINQ's Apply operator [29]. `MergeSort` takes multiple sorted input streams, and merge sorts them. `HashMergeGroups` and `SortMergeGroups` take multiple input streams and merge groups of the same key from the input streams. If all the input streams are sorted, Nectar chooses to use `SortMergeGroups`, which is streaming and more efficient. Otherwise, Nectar uses `HashMergeGroups`. The `MG` vertex in Figure 4 is an example of this group merge.

The technique of reusing materialized views in database systems addresses a similar problem. One important difference is that a database typically does not maintain views for multiple versions of a table, which would prevent it from reusing results computed on old incarnations of the table. For example, suppose we have a materialized view V on D. When D is changed to $D + D_1$, the view is also updated to V'. So for any fu-

ture computation on $D + D_2$, V is no longer available for use. In contrast, Nectar maintains both V and V', and automatically tries to reuse them for any computation, in particular the ones on $D + D_2$.

Cache Insertion Policy

We consider every prefix sub-expression of an expression to be a candidate for caching. Adding a cache entry incurs additional cost if the entry is not useful. It requires us to store the result of the computation on disk (instead of possibly pipelining the result to the next stage), incurring the additional disk IO and space overhead. Obviously it is not practical to cache everything. Nectar implements a simple strategy to determine what to cache.

First of all, Nectar always creates a cache entry for the final result of a computation as we get it for free: it does not involve a break of the computation pipeline and incurs no extra IO and space overhead.

For sub-expression candidates, we wish to cache them only when they are predicted to be useful in the future. However, determining the potential usefulness of a cache entry is generally difficult. So we base our cache insertion policy on heuristics. The caching decision is made in the following two phases.

First, when the rewriter rewrites an expression, it decides on the places in the expression to insert AddEntry calls. This is done using the usage statistics maintained by the cache server. The cache server keeps statistics for a sub-expression based on request history from clients. In particular, it records the number of times it has been looked up. On response to a cache lookup, this number is included in the return value. We insert an AddEntry call for an expression only when the number of lookups on it exceeds a predefined threshold.

Second, the decision made by the rewriter may still be wrong because of the lack of information about the saving of the computation. Information such as execution time and disk consumption are only available at run time. So the final insertion decision is made based on the runtime information of the execution of the sub-expression. Currently, we use a simple benefit function that is proportional to the execution time and inversely proportional to storage overhead. We add the cache entry when the benefit exceeds a threshold.

We also make our cache insertion policy adaptive to storage space pressure. When there is no pressure, we choose to cache more aggressively as long as it saves machine time. This strategy could increase the useless cache entries in the cache. But it is not a problem because it is addressed by Nectar's garbage collection, discussed further below.

3.2 Managing Derived Data

Derived datasets can take up a significant amount of storage space in a datacenter, and a large portion of it could be unused or seldom used. Nectar keeps track of the usage statistics of all derived datasets and deletes the ones of the least value. Recall that Nectar permanently stores the program of every derived dataset so that a deleted derived can be recreated by re-running its program.

Data Store for Derived Data

As mentioned before, Nectar stores all derived datasets in a data store inside a distributed, fault-tolerant file system. The actual location of a derived dataset is completely opaque to programmers. Accessing an existing derived dataset must go through the cache server. We expose a standard file interface with one important restriction: New derived datasets can only be created as results of computations.

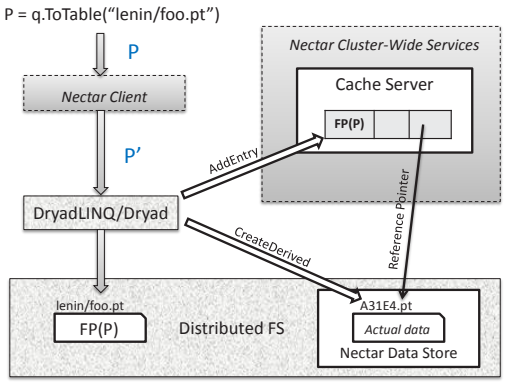


Figure 5: The creation of a derived dataset. The actual dataset is stored in the Nectar data store. The user file contains only the primary key of the cache entry associated with the derived.

Our scheme to achieve this is straightforward. Figure 5 shows the flow of creating a derived dataset by a computation and the relationship between the user file and the actual derived dataset. In the figure, `P` is a user program that writes its output to `lenin/foo.pt`. After applying transformations by Nectar and DryadLINQ, it is executed in the datacenter by Dryad. When the execution succeeds, the actual derived dataset is stored in the data store with a unique name generated by Nectar. A cache entry is created with the fingerprint of the program (`FP(P)`) as the primary key and the unique name as a field. The content of `lenin/foo.pt` just contains the primary key of the cache entry.

To access `lenin/foo.pt`, Nectar simply uses `FP(P)` to look up the cache to obtain the location of the actual derived dataset (`A31E4.pt`). The fact that all accesses go through the cache server allows us to keep

track of the usage history of every derived dataset and to implement automatic garbage collection for derived datasets based on their usage history.

Garbage Collection

When the available disk space falls below a threshold, the system automatically deletes derived datasets that are considered to be least useful in the future. This is achieved by a combination of the Nectar cache server and garbage collector.

A derived dataset is protected from garbage collection if it is referenced by any cache entry. So, the first step is to evict from the cache, entries that the cache server determines to have the least value.

The cache server uses information stored in the cache entries to do a cost-benefit analysis to determine the usefulness of the entries. For each cache entry, we keep track of the size of the resulting derived dataset (S), the elapsed time since it was last used (ΔT), the number of times (N) it has been used and the cumulative machine time (M) of the computation that created it. The cache server uses these values to compute the cost-to-benefit ratio

$$\text{CBRatio} = (S \times \Delta T)/(N \times M)$$

of each cache entry and deletes entries that have the largest ratios so that the cumulative space saving reaches a predefined threshold.

Freshly created cache entries do not contain information for us to compute a useful cost/benefit ratio. To give them a chance to demonstrate their usefulness, we exclude them from deletion by using a lease on each newly created cache entry.

The entire cache eviction operation is done in the background, concurrently with any other cache server operations. When the cache server completes its eviction, the garbage collector deletes all derived datasets not protected by a cache entry using a simple mark-and-sweep algorithm. Again, this is done in the background, concurrently with any other activities in the system.

Other operations can run concurrently with the garbage collector and create new cache entries and derived datasets. Derived datasets pointed to by cache entries (freshly created or otherwise) are not candidates for garbage collection. Notice however that freshly created derived datasets, which due to concurrency may not yet have a cache entry, also need to protected from garbage collection. We do this with a lease on the dataset.

With these leases in place, garbage collection is quite straightforward. We first compute the set of all derived datasets (ignoring the ones with unexpired leases) in our data store, exclude from it the set of all derived datasets referenced by cache entries, and treat the remaining as garbage.

Our system could mistakenly delete datasets that are subsequently requested, but these can be recreated by re-executing the appropriate program(s) from the program store. Programs are stored in binary form in the program store. A program is a complete Dryad job that can be submitted to the datacenter for execution. In particular, it includes the execution plan and all the application DLLs. We exclude all system DLLs, assuming that they are available on the datacenter machines. For a typical datacenter that runs 1000 jobs daily, our experience suggests it would take less than 1TB to store one year's program (excluding system DLLs) in uncompressed form. With compression, it should take up roughly a few hundreds of gigabytes of disk space, which is negligible even for a small datacenter.

4 Experimental Evaluation

We evaluate Nectar running on our 240-node research cluster as well as present analytic results of execution logs from 25 large production clusters that run jobs similar to those on our research cluster. We first present our analytic results.

4.1 Production Clusters

We use logs from 25 different clusters to evaluate the usefulness of Nectar. The logs consist of detailed execution statistics for 33182 jobs in these clusters for a recent 3-month period. For each job, the log has the source program and execution statistics such as computation time, bytes read and written and the actual time taken for every stage in a job. The log also gives information on the submission time, start time, end time, user information, and job status.

Programs from the production clusters work with massive datasets such as click logs and search logs. Programs are written in a language similar to DryadLINQ in that each program is a sequence of SQL-like queries [6]. A program is compiled into an expression tree with various stages and modeled as a DAG with vertices representing processes and edges representing data flows. The DAGs are executed on a Dryad cluster, just as in our Nectar managed cluster. Input data in these clusters is stored as append-only streams.

Benefits from Caching

We parse the execution logs to recreate a set of DAGs, one for each job. The root of the DAG represents the input to the job and a path through the DAG starting at the root represents a partial (i.e., a sub-) computation of the job. Identical DAGs from different jobs represent an opportunity to save part of the computation time of a later job by caching results from the earlier ones. We simulate

the effect of Nectar's caching on these DAGs to estimate cache hits.

Our results show that on average across all clusters, more than 35% of the jobs could benefit from caching. More than 30% of programs in 18 out of 25 clusters could have at least one cache hit, and there were even some clusters where 65% of programs could have cache hits.

The log contains detailed computation time information for each node in the DAG for a job. When there is a cache hit on a sub-computation of a job, we can therefore calculate the time saved by the cache hit. We show the result of this analysis in two different ways: Figure 6 shows the percentage of computing time saved and Table 1 shows the minimum number of hours of computation saved in each cluster.

Figure 6 shows that significant percentage of computation time can be saved in each cluster with Nectar. Most clusters can save a minimum of 20% to 40% of computation time and in some clusters the savings are up to 50%. Also, as an example, Table 1 shows a minimum of 7143 hours of computation per day can be saved using Nectar in Cluster C5. This is roughly equivalent to saying that about 300 machines in that cluster were doing wasteful computations all day that caching could eliminate. Across all 25 clusters, 35078 hours of computation per day can be saved, which is roughly equivalent to saving 1461 machines.

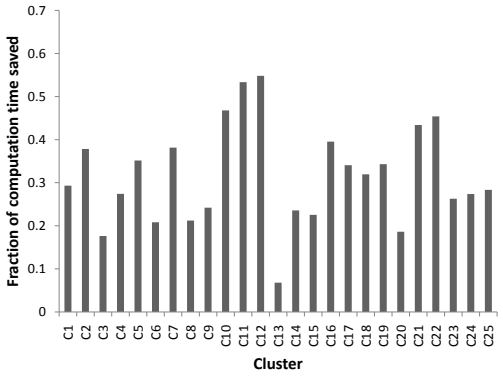


Figure 6: Fraction of compute time saved in each cluster

Ease of Program Development

Our analysis of the caching accounted for both sub-computation as well as incremental/sliding window hits. We noticed that the percentage of sliding window hits in some production clusters was minimal (under 5%). We investigated this further and noticed that many programmers explicitly structured their programs so that they can reuse a previous computation. This somewhat artificial structure makes their programs cumbersome, which can be alleviated by using Nectar.

Cluster	Computation Time Saved (hours/day)	Cluster	Computation Time Saved (hours/day)
C1	3898	C14	753
C2	2276	C15	755
C3	977	C16	2259
C4	1345	C17	3385
C5	7143	C18	528
C6	62	C19	4
C7	57	C20	415
C8	590	C21	606
C9	763	C22	2002
C10	2457	C23	1316
C11	1924	C24	291
C12	368	C25	58
C13	105		

Table 1: Minimum Computation Time Savings

There are anecdotes of system administrators manually running a common sub-expression on the daily input and explicitly notifying programmers to avoid each program performing the computation on its own and tying up cluster resources. Nectar automatically supports incremental computation and programmers do not need to code them explicitly. As discussed in Section 2, Nectar tries to produce the best possible query plan using the cached results, significantly reducing computation time, at the same time making it opaque to the user.

An unanticipated benefit of Nectar reported by our users on the research cluster was that it aids in debugging during program development. Programmers incrementally test and debug pieces of their code. With Nectar the debugging time significantly improved due to cache hits. We quantify the effect of this on the production clusters. We assumed that a program is a debugged version of another program if they had almost the same queries accessing the same source data and writing the same derived data, submitted by the same user and had the same program name.

Table 2 shows the amount of debugging time that can be saved by Nectar in the 90 day period. We present results for the first 12 clusters due to space constraints. Again, these are conservative estimates but shows substantial savings. For instance, in Cluster C1, a minimum of 3 hours of debugging time can be saved per day. Notice that this is actual elapsed time, i.e., each day 3 hours of computation on the cluster spent on debugging programs can be avoided with Nectar.

Cluster	Debugging Time Saved (hours)	Cluster	Debugging Time Saved (hours)
C1	270	C7	3
C2	211	C8	35
C3	24	C9	84
C4	101	C10	183
C5	94	C11	121
C6	8	C12	49

Table 2: Actual elapsed time saved on debugging in 90 days.

Managing Storage

Today, in datacenters, storage is manually managed.[1] We studied storage statistics in our 240-node research cluster that has been used by a significant number of users over the last 2 to 3 years. We crawled this cluster for derived objects and noted their last access times. Of the 109 TB of derived data, we discovered that about 50% (54.5 TB) was never accessed in the last 250 days. This shows that users often create derived datasets and after a point, forget about them, leaving them occupying unnecessary storage space.

We analyzed the production logs for the amount of derived datasets written. When calculating the storage occupied by these datasets, we assumed that if a new job writes to the same dataset as an old job, the dataset is overwritten. Figure 7 shows the growth of derived data storage in cluster C1. It show an approximately linear growth with the total storage occupied by datasets created in 90 days being 670 TB.

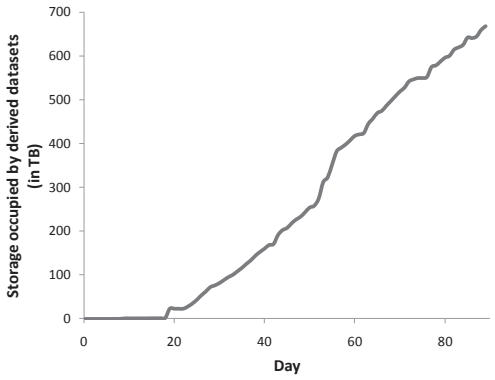


Figure 7: Growth of storage occupied by derived datasets in Cluster C1

[1]Nectar's motivation in automatically managing storage partly stems from the fact that we used to get periodic e-mail messages from the administrators of the production clusters requesting us to delete our derived objects to ease storage pressure in the cluster.

Cluster	Projected unreferenced derived data (in TB)
C1	2712
C5	368
C8	863
C13	995
C15	210

Table 3: Projected unreferenced data in 5 production clusters

Assuming similar trends in data access time in our local cluster and on the production clusters, Table 3 shows the projected space occupied by unreferenced derived datasets in 5 production clusters that showed a growth similar to cluster C1. Any object that has not been referenced in 250 days is deemed unreferenced. This result is obtained by extrapolating the amount of data written by jobs in 90 days to 2 years based on the storage growth curve and predicting that 50% of that storage will not be accessed in the last 250 days (based on the result from our local cluster). As we see, production clusters create a large amount of derived data, which if not properly managed can create significant storage pressure.

4.2 System Deployment Experience

Each machine in our 240-node research cluster has two dual-core 2.6GHz AMD Opteron 2218 HE CPUs, 16GB RAM, four 750GB SATA drives, and runs Windows Server 2003 operating system. We evaluate the comparative performance of several programs with Nectar turned on and off.

We use three datasets to evaluate the performance of Nectar:

WordDoc Dataset. The first dataset is a collection of Web documents. Each document contains a URL and its content (as a list of words). The data size is 987.4 GB . The dataset is randomly partitioned into 236 partitions. Each partition has two replicas in the distributed file system, evenly distributed on 240 machines.

ClickLog Dataset. The second dataset is a small sample from an anonymized click log of a commercial search engine collected over five consecutive days. The dataset is 160GB in size, randomly partitioned into 800 partitions, two replicas each, evenly distributed on 240 machines.

SkyServer Dataset. This database is taken from the Sloan Digital Sky Survey database [11]. It contains two data files: 11.8 and 41.8 GBytes of data. Both files were manually range-partitioned into 40 partitions using the same keys.

Sub-computation Evaluation

We have four programs: *WordAnalysis*, *TopWord*, *MostDoc*, and *TopRatio* that analyze the *WordDoc* dataset.

WordAnalysis parses the dataset to generate the number of occurrences of each word and the number of documents that it appears in. *TopWord* looks for the top ten most commonly used words in all documents. *MostDoc* looks for the top ten words appearing in the largest number of documents. *TopRatio* finds the percentage of occurrences of the top ten mostly used words among all words. All programs take the entire 987.4 GB dataset as input.

Program Name	Cumulative Time		Saving
	Nectar on	Nectar off	
TopWord	16.1m	21h44m	98.8%
MostDoc	17.5m	21h46m	98.6%
TopRatio	21.2m	43h30m	99.2%

Table 4: Saving by sharing a common sub-computation: Document analysis

With Nectar on, we can cache the results of executing the first program, which spends a huge amount of computation analyzing the list of documents to output an aggregated result of much smaller size (12.7 GB). The subsequent three programs share a sub-computation with the first program, which is satisfied from the cache. Table 4 shows the cumulative CPU time saved for the three programs. This behavior is not isolated, one of the programs that uses the *ClickLog* dataset shows a similar pattern; we do not report the results here for reasons of space.

Incremental Computation

We describe the performance of a program that studies query relevance by processing the *ClickLog* dataset. When users search a phrase at a search engine, they click the most relevant URLs returned in the search results. Monitoring the URLs that are clicked the most for each search phrase is important to understand query relevance. The input to the query relevance program is the set of all click logs collected so far, which increases each day, because a new log is appended daily to the dataset. This program is an example where the initial dataset is large, but the incremental updates are small.

Table 5 shows the cumulative CPU time with Nectar on and off, the size of datasets and incremental updates each day. We see that the total size of input data increases each day, while the computation resource used daily increases much slower when Nectar is on. We observed similar performance results for another program that calculates the number of active users, who are those that clicked at least one search result in the past three days. These results are not reported here for reasons of space.

	Data Size(GB)		Time (m)		Saving
	Total	Update	On	Off	
Day3	68.20	40.50	93.0	107.5	13.49%
Day4	111.25	43.05	112.9	194.0	41.80%
Day5	152.19	40.94	164.6	325.8	49.66%

Table 5: Cumulative machine time savings for incremental computation.

Debugging Experience: Sky Server

Here we demonstrate how Nectar saves program development time by shortening the debugging cycle. We select the most time-consuming query (Q18) from the Sloan Digital Sky Survey database [11]. The query identifies a gravitational lens effect by comparing the locations and colors of stars in a large astronomical table, using a three-way Join over two input tables containing 11.8 GBytes and 41.8 GBytes of data, respectively. The query is composed of four steps, each of which is debugged separately. When debugging the query, the first step failed and the programmer modified the code. Within a couple of tries, the first step succeeded, and execution continued to the second step, which failed, and so on.

Table 6 shows the average savings in cumulative time as each step is successively debugged with Nectar. Towards the end of the program, Nectar saves as much 88% of the time.

	Cumulative Time		Saving
	Nectar on	Nectar off	
Step 1	47.4m	47.4m	0%
Steps 1–2	26.5m	62.5m	58%
Steps 1–3	35.5m	122.7m	71%
Steps 1–4	15.0m	129.3m	88%

Table 6: Debugging: SkyServer cumulative time

5 Related Work

Our overall system architecture is inspired by the Vesta system [15]. Many high-level concepts and techniques (e.g., the notion of primary and derived data) are directly taken from Vesta. However, because of the difference in application domains, the actual design and implementation of the main system components such as caching and program rewriting are radically different.

Many aspects of query rewriting and caching in our work are closely related to incremental view maintenance and materialized views in the database literature [2, 5, 13, 19]. However, there are some important differences as discussed in Section 3.1. Also, we are not aware of the implementation of these ideas in systems

at the scale we describe in this paper. Incremental view maintenance is concerned with the problem of updating the materialized views incrementally (and consistently) when data base tables are subjected to random updates. Nectar is simpler in that we only consider append-only updates. On the other hand, Nectar is more challenging because we must deal with user-defined functions written in a general-purpose programming language. Many of the sophisticated view reuses given in [13] require analysis of the SQL expressions that is difficult to do in the presence of user-defined functions, which are common in our environment.

With the wide adoption of distributed execution platforms like Dryad/DryadLINQ, MapReduce/Sawzall, Hadoop/Pig [18, 29, 7, 25, 12, 24], recent work has investigated job patterns and resource utilization in data centers [1, 14, 22, 23, 26]. These investigation of real work loads have revealed a vast amount of wastage in datacenters due to redundant computations, which is consistent with our findings from logs of a number of production clusters.

DryadInc [26] represented our early attempt to eliminate redundant computations via caching, even before we started on the DryadLINQ project. The caching approach is quite similar to Nectar. However, it works at the level of Dryad dataflow graph, which is too general and too low-level for the system we wanted to build.

The two systems that are most related to Nectar are the stateful bulk processing system described by Logothetis et al. [22] and Comet [14]. These systems mainly focus on addressing the important problem of incremental computation, which is also one of the problems Nectar is designed to address. However, Nectar is a much more ambitious system, attempting to provide a comprehensive solution to the problem of automatic management of data and computation in a datacenter.

As a design principle, Nectar is designed to be transparent to the users. The stateful bulk processing system takes a different approach by introducing new primitives and hence makes *state* explicit in the programming model. It would be interesting to understand the tradeoffs in terms of performance and ease of programming.

Comet, also built on top of Dryad and DryadLINQ, also attempted to address the sub-computation problem by co-scheduling multiple programs with common subcomputations to execute together. There are two interesting issues raised by the paper. First, when multiple programs are involved in caching, it is difficult to determine if two code segments from different programs are identical. This is particularly hard in the presence of user-defined functions, which is very common in the kind of DryadLINQ programs targeted by both Comet and Nectar. It is unclear how this determination is made in Comet. Nectar addresses this problem by building a sophisticated static program analyzer that allows us to compute the dependency of user-defined code. Second, co-scheduling in Comet requires submissions of multiple programs with the same timestamp. It is therefore not useful in all scenarios. Nectar instead shares subcomputations across multiple jobs executed at different times by using a datacenter-wide, persistent cache service.

Caching function calls in a functional programming language is well studied in the literature [15, 21, 27]. Memoization avoids re-computing the same function calls by caching the result of past invocations. Caching in Nectar can be viewed as function caching in the context of large-scale distributed computing.

6 Discussion and Conclusions

In this paper, we described Nectar, a system that automates the management of data and computation in datacenters. The system has been deployed on a 240-node research cluster, and has been in use by a small number of developers. Feedback has been quite positive. One very popular comment from our users is that the system makes program debugging much more interactive and fun. Most of us, the Nectar developers, use Nectar to develop Nectar on a daily basis, and found a big increase in our productivity.

To validate the effectiveness of Nectar, we performed a systematic analysis of computation logs from 25 production clusters. As reported in Section 4, we have seen huge potential value in using Nectar to manage the computation and data in a large datacenter. Our next step is to work on transferring Nectar to Microsoft production datacenters.

Nectar is a complex distributed systems with multiple interacting policies. Devising the right policies and fine-tuning their parameters to find the right trade-offs is essential to make the system work in practice. Our evaluation of these tradeoffs has been limited, but we are actively working on this topic. We hope we will continue to learn a great deal with the ongoing deployment of Nectar on our 240-node research cluster.

One aspect of Nectar that we have not explored is that it maintains the provenance of all the derived datasets in the datacenter. Many important questions about data provenance could be answered by querying the Nectar cache service. We plan to investigate this further in future work.

What Nectar essentially does is to unify computation and data, treating them interchangeably by maintaining the dependency between them. This allows us to greatly improve the datacenter management and resource utilization. We believe that it represents a significant step forward in automating datacenter computing.

Acknowledgments

We would like to thank Dennis Fetterly and Maya Haridasan for their help with TidyFS. We would also like to thank Martín Abadi, Surajit Chaudhuri, Yanlei Diao, Michael Isard, Frank McSherry, Vivek Narasayya, Doug Terry, and Fang Yu for many helpful comments. Thanks also to the OSDI review committee and our shepherd Pei Cao for their very useful feedback.

References

[1] AGRAWAL, P., KIFER, D., AND OLSTON, C. Scheduling shared scans of large data files. *Proc. VLDB Endow. 1*, 1 (2008), 958–969.

[2] AGRAWAL, S., CHAUDHURI, S., AND NARASAYYA, V. R. Automated selection of materialized views and indexes in SQL databases. In *VLDB* (2000), pp. 496–505.

[3] ALVARO, P., CONDIE, T., CONWAY, N., ELMELEEGY, K., HELLERSTEIN, J. M., AND SEARS, R. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems* (2010), pp. 223–236.

[4] BRODER, A. Z. Some applications of Rabinś fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science* (1993), Springer-Verlag, pp. 143–152.

[5] CERI, S., AND WIDOM, J. Deriving production rules for incremental view maintenance. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases* (1991), pp. 577–589.

[6] CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (2008), 1265–1276.

[7] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (2008), 107–113.

[8] FETTERLY, D., HARIDASAN, M., ISARD, M., AND SUNDARARAMAN, S. TidyFS: A simple and small distributed filesystem. Tech. Rep. MSR-TR-2010-124, Microsoft Research, October 2010.

[9] GOLUMBIC, M. C. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol. 57)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2004.

[10] GRAY, J., CHAUDHURI, S., BOSWORTH, A., LAYMAN, A., REICHART, D., VENKATRAO, M., PELLOW, F., AND PIRAHESH, H. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery 1*, 1 (1997).

[11] GRAY, J., SZALAY, A., THAKAR, A., KUNSZT, P., STOUGHTON, C., SLUTZ, D., AND VANDENBERG, J. Data mining the SDSS SkyServer database. In *Distributed Data and Structures 4: Records of the 4th International Meeting* (Paris, France, March 2002), Carleton Scientific, pp. 189–210. Also available as MSR-TR-2002-01.

[12] The Hadoop project. `http://hadoop.apache.org/`.

[13] HALEVY, A. Y. Answering Queries Using Views: A Survey. *VLDB J. 10*, 4 (2001), 270–294.

[14] HE, B., YANG, M., GUO, Z., CHEN, R., SU, B., LIN, W., AND ZHOU, L. Comet: batched stream processing for data intensive distributed computing. In *ACM Symposium on Cloud Computing (SOCC)* (2010), pp. 63–74.

[15] HEYDON, A., LEVIN, R., MANN, T., AND YU, Y. *Software Configuration Management Using Vesta*. Springer-Verlag, 2006.

[16] HEYDON, A., LEVIN, R., AND YU, Y. Caching function calls using precise dependencies. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (New York, NY, USA, 2000), ACM, pp. 311–320.

[17] The HIVE project. `http://hadoop.apache.org/hive/`.

[18] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (2007), pp. 59–72.

[19] LEE, K. Y., SON, J. H., AND KIM, M. H. Efficient incremental view maintenance in data warehouses. In *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management* (2001), pp. 349–356.

[20] The LINQ project. `http://msdn.microsoft.com/netframework/future/linq/`.

[21] LIU, Y. A., STOLLER, S. D., AND TEITELBAUM, T. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst. 20*, 3 (1998), 546–585.

[22] LOGOTHETIS, D., OLSTON, C., REED, B., WEBB, K., AND YOCUM, K. Stateful bulk processing for incremental algorithms. In *ACM Symposium on Cloud Computing (SOCC)* (2010).

[23] OLSTON, C., REED, B., SILBERSTEIN, A., AND SRIVASTAVA, U. Automatic optimization of parallel dataflow programs. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference* (2008), pp. 267–273.

[24] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), pp. 1099–1110.

[25] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming 13*, 4 (2005).

[26] POPA, L., BUDIU, M., YU, Y., AND ISARD, M. DryadInc: Reusing work in large-scale computations. In *Workshop on Hot Topics in Cloud Computing (HotCloud)* (San Diego, CA, June 15 2009).

[27] PUGH, W., AND TEITELBAUM, T. Incremental computation via function caching. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1989), pp. 315–328.

[28] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), pp. 247–260.

[29] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)* (2008), pp. 1–14.

Intrusion Recovery Using Selective Re-execution

Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek
MIT CSAIL

ABSTRACT

RETRO repairs a desktop or server after an adversary compromises it, by undoing the adversary's changes while preserving legitimate user actions, with minimal user involvement. During normal operation, RETRO records an *action history graph*, which is a detailed dependency graph describing the system's execution. RETRO uses *refinement* to describe graph objects and actions at multiple levels of abstraction, which allows for precise dependencies. During repair, RETRO uses the action history graph to undo an unwanted action and its indirect effects by first rolling back its direct effects, and then re-executing legitimate actions that were influenced by that change. To minimize user involvement and re-execution, RETRO uses *predicates* to *selectively re-execute* only actions that were semantically affected by the adversary's changes, and uses *compensating actions* to handle external effects.

An evaluation of a prototype of RETRO for Linux with 2 real-world attacks, 2 synthesized challenge attacks, and 6 attacks from previous work, shows that RETRO can often repair the system without user involvement, and avoids false positives and negatives from previous solutions. These benefits come at the cost of 35–127% in execution time overhead and of 4–150 GB of log space per day, depending on the workload. For example, a HotCRP paper submission web site incurs 35% slowdown and generates 4 GB of logs per day under the workload from 30 minutes prior to the SOSP 2007 deadline.

1 INTRODUCTION

Despite our best efforts to build secure computer systems, intrusions are nearly unavoidable in practice. When faced with an intrusion, a user is typically forced to reinstall their system from scratch, and to manually recover any documents and settings they might have had. Even if the user diligently makes a complete backup of their system every day, recovering from the attack requires rolling back to the most recent backup before the attack, thereby losing any changes made since then. Since many adversaries go to great lengths to prevent the compromise from being discovered, it can take days or weeks for a user to discover that their machine has been broken into, resulting in a loss of all user work from that period of time.

This paper presents RETRO, a system for retroactively undoing past attacks and their indirect effects on a single machine. With RETRO, an administrator specifies offending actions from the past, such as a TCP connection or an HTTP request from an adversary, that they want to undo. RETRO then repairs the system's state (the file system) by selectively undoing the offending actions—that is, constructing a new system state, as if the offending actions never took place, but all legitimate actions remained. Thus, by selectively undoing the adversary's changes while preserving user data, RETRO makes intrusion recovery more practical.

To illustrate the challenges facing RETRO, consider the following attack, which we will use as a running example in this paper. Eve, an evil adversary, compromises a Linux machine, and obtains a root shell. To mask her trail, she removes the last hour's entries from the system log. She then creates several backdoors into the system, including a new account for `eve`, and a PHP script that allows her to execute arbitrary commands via HTTP. Eve then uses one of these backdoors to download and install a botnet client. To ensure continued control of the machine, Eve adds a line to the `/usr/bin/texi2pdf` shell script (a wrapper for LaTeX) to restart her bot. In the meantime, legitimate users log in, invoke their own PHP scripts, use `texi2pdf`, and root adds new legitimate users.

To undo attacks, RETRO provides a system-wide architecture for recording actions, causes, and effects in order to identify all the downstream effects of a compromise. The key challenge is that a compromise in the past may have effects on subsequent legitimate actions, especially if the administrator discovers an attack long after it occurred. RETRO must sort out this entanglement automatically and efficiently. In our running example, Eve's changes to the password file and to `texi2pdf` are entangled with legitimate actions that modified or accessed the password file, or used `texi2pdf`. If legitimate users ran `texi2pdf`, their output depended on Eve's actions, and so did any programs that used that output in turn.

As described in §2, most previous systems require user input to disentangle such actions. Typical previous solutions are good at detecting a compromise and allow a user to roll the system back to a check point before the compromise, but then ask the user to incorporate legitimate changes from after the compromise manually; this can be quite onerous if the attack has happened a long time ago. Some solutions reduce the amount of manual work for special cases (e.g., known viruses). The most recent general solution for reducing user assistance (Taser [17]) incurs many false positives (undoing legitimate actions),

or, after white-listing some actions to minimize false positives, it incurs false negatives (missing parts of the attack).

How can RETRO disentangle unwanted actions from legitimate operations, and undo all effects of the adversary's actions that happened in the past, while preserving every legitimate action? RETRO addresses these challenges with four ideas:

First, RETRO models the entire system using a new form of a dependency graph, which we call an *action history graph*. Like any dependency graph, the action history graph represents objects in the system (such as files and processes), and the dependencies between those objects (corresponding to actions such as a process reading a file). To record precise dependencies, the action history graph supports *refinement*, that is, representing the same object or action at multiple levels of abstraction. For example, a directory inode can be refined to expose individual file names in that directory, and a process can be refined into function calls and system calls. The action history graph also captures the semantics of each dependency (e.g., the arguments and return values of an action).

Second, RETRO *re-executes* actions in the graph, such as system calls or process invocations, that were influenced by the offending changes. For example, undoing undesirable actions may indirectly change the inputs of later actions, and thus these actions must be re-executed with their repaired inputs.

Third, RETRO uses *predicates* to do *selective re-execution* of just the actions whose dependencies are semantically different after repair, thereby minimizing cascading re-execution. For example, if Eve modified some file, and that file was later read by process P, we may be able to avoid re-executing P if the part of the file accessed by P is the same before and after repair.

Finally, to selectively re-execute existing applications, RETRO uses *shepherded re-execution* to monitor the re-execution of processes (§5.2.3), and stops re-execution when the process state converges to the original execution (such as when a process issues an identical `exec` call).

Using a prototype of RETRO for Linux, we show that RETRO can recover from both real-world and synthetic attacks, including our running example, while preserving legitimate user changes. Out of ten experiment scenarios, six required no user input to repair, two required user confirmation that a conflicting login session belonged to the attacker, and two required the user to manually redo affected operations. We also show that RETRO's ideas of refinement, shepherded re-execution, and predicates are key to repairing precisely the files affected by the attack, and to minimizing user involvement. A performance evaluation shows that, for extreme workloads that issue many system calls (such as continuously recompiling the Linux kernel), RETRO imposes a 89–127% runtime overhead and requires 100–150 GB of log space per day. For a more realistic application, such as a HotCRP [23] conference submission site, these costs are 35% and 4 GB per day, respectively. RETRO's runtime cost can be reduced by using additional cores, amounting to 0% for HotCRP when one core is dedicated to RETRO.

The rest of the paper is organized as follows. The next section compares RETRO with related work. §3 presents an overview of RETRO's architecture and workflow. §4 discusses RETRO's action history graph in detail, and §5 describes RETRO's repair managers. Our prototype implementation is described in §6, and §7 evaluates the effectiveness and performance of RETRO. Finally, §8 discusses the limitations and future work, and §9 concludes.

2 RELATED WORK

This section relates RETRO to industrial and academic solutions for recovery after a compromise, and prior techniques that RETRO builds on.

2.1 Repair solutions

One line of industrial solutions is anti-virus tools, which can revert changes made by common malware, such as Windows registry keys and files comprising a known virus. For example, tools such as [34] can generate remediation procedures for a given piece of malware. While such techniques work for known malware that behaves in predictable ways, they incur both false positives and false negatives, especially for new or unpredictable malware, and may not be able to recover from attacks where some information is lost, such as file deletions or overwrites. They also cannot repair changes that were a side-effect of the attack, such as changes made by a trojaned program, or changes made by an interactive adversary, whereas RETRO can undo such changes.

Another line of industrial solutions is systems that help users roll back unwanted changes to system state. These solutions include Windows System Restore [18], Windows Driver Rollback [30], Time Machine [4], and numerous backup tools. These tools perform coarse-grained recovery, and require the user to identify what files were affected. RETRO uses the action history graph to track down *all* effects of an attack, repairs *precisely* those changes, and repairs all *side-effects* of the attack, without requiring the user to guess what files were affected.

A final line of popular solutions is using virtual machines as a form of whole-system backup. Using ReVirt [14] or Moka5 [11, 31], an administrator can roll back to a checkpoint before an attack, losing both the attacker's changes and any legitimate changes since that point. One could imagine a system that replays recorded legitimate network packets to the virtual machine to reapply legitimate changes. However, if there are even subtle dependencies between omitted and replayed packets, the replayed packets will result in conflicts or external

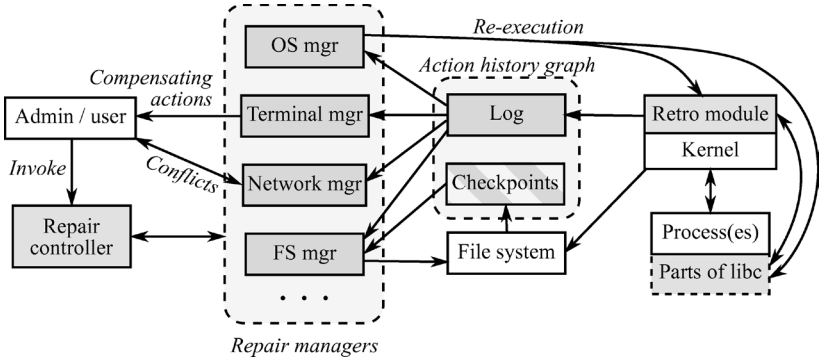


Figure 1: Overview of RETRO's architecture, including major components and their interactions. Shading indicates components introduced by RETRO. Striped shading of checkpoints indicates that RETRO reuses existing file system snapshots when available.

dependencies, requiring user input to proceed. By recording dependencies and re-executing actions at many levels of abstraction using refinement, RETRO avoids such conflicts and can preserve legitimate changes without user input.

Academic research has tried to improve over the industrial solutions by attempting to make solutions more automatic. Brown's undoable email store [10] shows how an email server can recover from operator mistakes, by turning all operations into *verbs*, such as SMTP or IMAP commands. Unlike RETRO, Brown's approach is limited to recovering from accidental operator mistakes. As a result, it cannot deal with an adversary that goes outside of the *verb* model and takes advantage of a vulnerability in the IMAP server software, or guesses root's password to log in via ssh. Moreover, it cannot recover from actions that had system-wide effects spanning multiple applications, files, and processes.

The closest related work to RETRO is Taser [17], which uses taint tracking to find files affected by a past attack. Taser suffers from false positives, erroneously rolling back hundreds or thousands of files. To prevent false positives, Taser uses a white-list to ignore taint for some nodes or edges. This causes false negatives, so an attacker can bypass Taser altogether. While extensions of Taser catch some classes of attacks missed due to false negatives [40], RETRO has no need for white-listing. RETRO recovers from all attacks presented in the Taser paper with no false positives or false negatives. RETRO avoids Taser's limitations by using a design based on the action history graph, and techniques such as predicates and re-execution, as opposed to Taser's taint propagation.

Polygraph [29] uses taint tracking to recover from compromised devices in a data replication system, and incurs false positives like Taser. Unlike RETRO, Polygraph can recover from compromises in a distributed system.

2.2 Related techniques

The use of dependency information for security has been widely explored in many contexts, including information flow control [25, 45], taint tracking [44], data provenance [9], forensics [21], system integrity [8], and so on. A key difference in RETRO's action history graph is the use of exact dependency data to decide whether a dependency has semantically changed at repair time.

RETRO assumes that intrusion detection and analysis tools, such as [7, 12, 14, 15, 19–22, 24, 40, 43], detect attacks and pinpoint attack edges. RETRO's intrusion detection is based on BackTracker [21]. A difference is that RETRO's action history graph records more information than BackTracker, which RETRO needs for repair (but doesn't use yet for detection).

Transactions [33, 36] help revert unwanted changes before commit, whereas RETRO can selectively undo "committed" actions. Database systems use compensating transactions to revert committed transactions, including malicious transactions [3, 27]; RETRO similarly uses compensating actions to deal with externally-visible changes.

3 OVERVIEW

RETRO consists of several components, as shown in Figure 1. During normal execution, RETRO's kernel module records a log of system execution, and creates periodic checkpoints of file system state. When the system administrator notices a problem, he or she uses RETRO to track down the initial intrusion point. Given an intrusion point, RETRO reverts the intrusion, and repairs the rest of the system state, relying on the system administrator to resolve any conflicts (e.g., both the adversary and a legitimate user modified the same line of the password file). The rest of this section describes these phases of operation in more detail, and outlines the assumptions made by RETRO about the system and the adversary.

Normal execution. As the computer executes, RETRO must record sufficient information to be able to revert the effects of an attack. To this end, RETRO records periodic checkpoints of persistent state (the file system), so that it can later roll back to a checkpoint. RETRO does not require any specialized format for its file system

checkpoints; if the file system already creates periodic snapshots, such as [26, 32, 37, 38], RETRO can simply use these snapshots, and requires no checkpointing of its own. In addition to rollback, RETRO must be able to re-execute affected computations. To this end, RETRO logs actions executed over time, along with their dependencies. The resulting checkpoints and actions comprise RETRO's *action history graph*, such as the one shown in Figure 2.

The action history graph consists of two kinds of objects: *data objects*, such as files, and *actor objects*, such as processes. Each object has a set of checkpoints, representing a copy of its state at different points in time. Each actor object additionally consists of a set of *actions*, representing the execution of that actor over some period of time. Each action has dependencies from and to other objects in the graph, representing the objects accessed and modified by that action. Actions and checkpoints of adjacent objects are ordered with respect to each other, in the order in which they occurred.[1]

RETRO stores the action history graph in a series of log files over time. When RETRO needs more space for new log files, it garbage-collects older log files (by deleting them). Log files are only useful to RETRO in conjunction with a checkpoint that precedes the log files, so log files with no preceding checkpoint can be garbage-collected. In practice, this means that RETRO keeps checkpoints for at least as long as the log files. By design, RETRO cannot recover from an intrusion whose log files have been garbage collected; thus, the amount of log space allocated to logs and checkpoints controls RETRO's recovery "horizon". For example, a web server running the HotCRP paper review software [23] logs 4 GB of data per day, so if the administrator dedicates a 2 TB disk ($100) to RETRO, he or she can recover from attacks within the past year, although these numbers strongly depend on the application.

Intrusion detection. At some point after an adversary compromises the system, the system administrator learns of the intrusion, perhaps with the help of an intrusion detection system. To repair from the intrusion, the system administrator must first track down the initial intrusion point, such as the adversary's network connection, or a user accidentally running a malware binary. RETRO provides a tool similar to BackTracker [21] that helps the administrator find the intrusion point, starting from the observed symptoms, by leveraging RETRO's action history graph. In the rest of this paper, we assume that an intrusion detection system exists, and we do not describe our BackTracker-like tool in any more detail.

Repair. Once the administrator finds the intrusion point, he or she reboots the system, to discard non-persistent state, and invokes RETRO's repair controller, specifying the name of the intrusion point determined in the previous step.[2] The repair controller undoes the offending action, A, by rolling back objects modified by A to a previous checkpoint, and replacing A with a no-op in the action history graph. Then, using the action history graph, the controller determines which other actions were potentially influenced by A (e.g., the values of their arguments changed), rolls back the objects they depend on (e.g., their arguments) to a previous checkpoint, re-executes those actions in their corrected environment (e.g., with the rolled-back arguments), and then repeats the process for actions that the re-executed actions may have influenced. This process will also undo subsequent actions by the adversary, since the action that initially caused them, A, has been reverted. Thus, after repair, the system will contain the effects of all legitimate actions since the compromise, but none of the effects of the attack.

To minimize re-execution and to avoid potential conflicts, the repair controller checks whether the inputs to each action are semantically equivalent to the inputs during original execution, and skips re-execution in that case. In our running example, if Alice's `sshd` process reads a password file that Eve modified, it might not be necessary to re-execute `sshd` if its execution only depended on Alice's password entry, and Eve did not change that entry. If Alice's `sshd` later changed her password entry, then this change will not result in a conflict during repair because the repair controller will determine that her change to the password file could not have been influenced by Eve.

RETRO's repair controller must manipulate many kinds of objects (e.g., files, directories, processes, etc.) and re-execute many types of actions (e.g., system calls and function calls) during repair. To ensure that RETRO's design is extensible, RETRO's action history graph provides a well-defined API between the repair controller and individual graph objects and actions. Using this API, the repair controller implements a generic repair algorithm, and interacts with the graph through individual *repair managers* associated with each object and action in the action history graph. Each repair manager, in turn, tracks the state associated with their respective object or action, implements object/action-specific operations during repair, and efficiently stores and accesses the on-disk state, logs, and checkpoints.

External dependencies. During repair, RETRO may discover that changes made by the adversary were externally visible. RETRO relies on compensating actions to deal with external dependencies where possible. For example, if a user's terminal output changes, RETRO sends

[1] For simplicity, our prototype globally orders all checkpoints and actions for all objects.

[2] Each object and action in the action history graph has a unique name, as described in §5.

a diff between the old and new terminal sessions to the user in question.

In some cases, RETRO does not have a compensating action to apply. If Eve, from our running example, connected to her botnet client over the network, RETRO would not be able to re-execute the connection during repair (the connection will be refused since the botnet will no longer be running). When such a situation arises, RETRO's repair controller pauses re-execution and asks the administrator to manually re-execute the appropriate action. In the case of Eve's connection, the administrator can safely do nothing and tell the repair controller to resume.

Assumptions. RETRO makes three significant assumptions. First, RETRO assumes that the system administrator detects intrusions in a timely manner, that is, before the relevant logs are garbage-collected. An adversary that is aware of RETRO could compromise the system and then try to avoid detection, by minimizing any activity until RETRO garbage-collects the logs from the initial intrusion. If the initial intrusion is not detected in time, the administrator will not be able to revert it directly, but this strategy would greatly slow down attackers. Moreover, the administrator may be able to revert subsequent actions by the adversary that leveraged the initial intrusion to cause subsequent notable activity.

Second, RETRO assumes that the administrator promptly detects any intrusions with wide-ranging effects on the execution of the entire system. If such intrusions persist for a long time, RETRO will require re-execution of large parts of the system, potentially incurring many conflicts and requiring significant user input. However, we believe this assumption is often reasonable, since the goal of many adversaries is to remain undetected for as long as possible (e.g., to send more spam, or to build up a large botnet), and making pervasive changes to the system increases the risk of detection.

Third, for this paper, we assume that the adversary compromises a computer system through user-level services. The adversary may install new programs, add backdoors to existing programs, modify persistent state and configuration files, and so on, but we assume the adversary doesn't tamper with the kernel, file system, checkpoints, or logs. RETRO's techniques rely on a detailed understanding of operating system objects, and our assumptions allow RETRO to trust the kernel state of these objects. We rely on existing techniques for hardening the kernel, such as [16, 28, 39, 41], to achieve this goal in practice.

4 ACTION HISTORY GRAPH

RETRO's design centers around the *action history graph*, which represents the execution of the entire system over time. The action history graph must address four requirements in order to disentangle attacker actions from legitimate operations. First, it must operate *system-wide*, capturing all dependencies and actions, to ensure that RETRO can detect and repair all effects of an intrusion. Second, the graph must support *fine-grained re-execution* of just the actions affected by the intrusion, without having to re-execute unaffected actions. Third, the graph must be able to *disambiguate attack actions* from legitimate operations whenever possible, without introducing false dependencies. Finally, recording and accessing the action history graph must be *efficient*, to reduce both runtime overheads and repair time. The rest of this section describes the design of RETRO's action history graph.

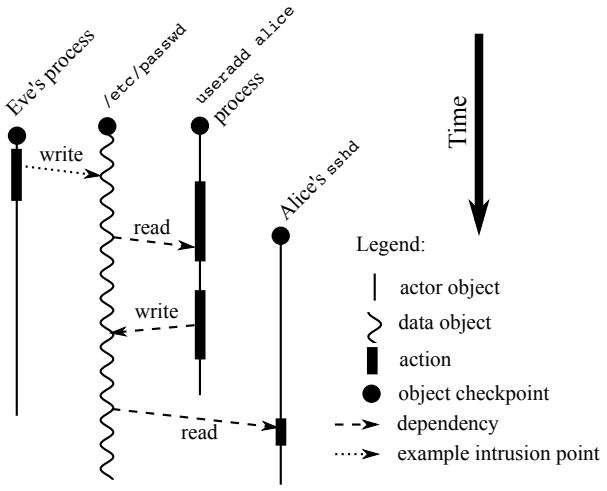


Figure 2: A simplified view of the action history graph depicting Eve's attack in our running example. In this graph, attacker Eve adds an account for herself to `/etc/passwd`, after which root adds an account for Alice, and Alice logs in via ssh. As an example, we consider Eve's write to the password file to be the attack action, although in reality, the attack action would likely be the network connection that spawned Eve's process in the first place. Not shown are intermediate data objects, and system call actors, described in §4.3 and Figure 4.

4.1 Repair using the action history graph

RETRO represents an attack as a set of *attack actions*. For example, an attack action can be a process reading data from the attacker's TCP connection, a user inadvertently running malware, or an offending file write. Given a set of attack actions, RETRO repairs the system in two steps, as follows.

First, RETRO replaces the attack actions with benign actions in the action history graph. For example, if the attack action was a process reading a malicious request from the attacker's TCP connection, RETRO removes the request data, as if the attacker never sent any data on that connection. If the attack action was a user accidentally running malware, RETRO changes the user's `exec` system call to run `/bin/true` instead of the malware binary. Finally, if the attack action was an unwanted write to a

Function or variable		Semantics
$set\langle checkpt\rangle$	object.checkpts	Set of available checkpoints for this object.
$void$	object.rollback(c)	Roll back this object to checkpoint c.
$set\langle action\rangle$	actor_object.actions	Set of actions that comprise this actor object.
$set\langle action\rangle$	data_object.readers	Set of actions that have a dependency from this data object.
$set\langle action\rangle$	data_object.writers	Set of actions that have a dependency to this data object.
$set\langle data\_object\rangle$	data_object.parts	Set of data objects whose state is part of this data object.
$actor\_object$	action.actor	Actor containing this action.
$set\langle data\_object\rangle$	action.inputs	Set of data objects that this action depends on.
$set\langle data\_object\rangle$	action.outputs	Set of data objects that depend on this action.
$bool$	action.equiv()	Check whether any inputs of this action have changed.
$bool$	action.connect()	Add dependencies for new inputs and outputs, based on new inputs.
$void$	action.redo()	Re-execute this action, updating output objects.

Figure 3: Object (top) and action (bottom) repair manager API.

file, as in Figure 2, RETRO replaces the action with a zero-byte write. RETRO includes a handful of such benign actions used to neutralize intrusion points found by the administrator.

Second, RETRO repairs the system state to reflect the above changes, by iteratively re-executing affected actions, starting with the benign replacements of the attack actions themselves. Prior to re-executing an action, RETRO must roll back all input and output objects of that action, as well as the actor itself, to an earlier checkpoint. For example, in Figure 2, RETRO rolls back the output of the attack action—namely, the password file object—to its earlier checkpoint.

RETRO then considers all actions with dependencies to or from the objects in question, according to their time order. Actions with dependencies *to* the object in question are re-executed, to reconstruct the object. For actions with dependencies *from* the object in question, RETRO checks whether their inputs are semantically equivalent to their inputs during original execution. If the inputs are different, such as the `useradd` command reading the modified password file in Figure 2, the action will be re-executed, following the same process as above. On the other hand, if the inputs are semantically equivalent, RETRO skips re-execution, avoiding the repair cascade. For example, re-executing `sshd` may be unnecessary, if the password file entry accessed by `sshd` is the same before and after repair. We will describe shortly how RETRO determines this (in §4.4 and Figure 5).

4.2 Graph API

As described above, repairing the system requires three functions: rolling back objects to a checkpoint, re-executing actions, and checking an action's input dependencies for semantic equivalence. To support different types of objects and actions in a system-wide action history graph, RETRO delegates these tasks, as well as tracking the graph structure itself, to *repair managers* associated with each object and action in the graph.

A manager consists of two halves: a runtime half, responsible for recording logs and checkpoints during normal execution, and a repair-time half, responsible for repairing the system state once the system administrator invokes RETRO to repair an intrusion. The runtime half has no pre-defined API, and needs to only synchronize its log and checkpoint format with the repair-time half. On the other hand, the repair-time half has a well-defined API, shown in Figure 3.

Object manager. During normal execution, object managers are responsible for making periodic checkpoints of objects. For example, the file system manager takes snapshots of files, such as a copy of `/etc/passwd` in Figure 2. Process objects also have checkpoints in the graph, although in our prototype, the only supported process checkpoint is the initial state of a process immediately prior to `exec`.

During repair, an object manager is responsible for maintaining the state represented by its object. For persistent objects, the manager uses the on-disk state, such as the actual file for a file object. For ephemeral objects, such as processes or pipes, the manager keeps a temporary in-memory representation to help action managers redo actions and check predicates, as we describe in §5.

An object manager provides one main procedure invoked during repair, $o.rollback(v)$, which rolls back object o's state to checkpoint v. For a file object, this means restoring the on-disk file from snapshot v. For a process, this means constructing an initial, paused process in preparation for redoing `exec`, as we will discuss in §5.2.3; since there is only one kind of process checkpoint, v is not used. If the object was last checkpointed long ago, RETRO will need to re-execute all subsequent actions that modified the data object, or that comprise the actor object.

Action manager. During normal execution, action managers are responsible for recording all actions executed by actors in the system. For each action, the manager records enough information to re-execute the same action at repair time, as well as to check whether the inputs are

semantically equivalent (e.g., by recording the data read from a file).

At repair time, an action manager provides three procedures. First, $a.redo()$ re-executes action a, reading new data from a's input objects and modifying the state of a's output objects. For example, redoing a file write action modifies the corresponding file in the file system; if the action was not otherwise modified, this would write the same data to the same offset as during original execution. Second, $a.equiv()$ checks whether a's inputs have semantically changed since the original execution. For instance, *equiv* on a file read action checks whether the file contains the same data at the same offset (and, therefore, whether the read call would return the same data). Finally, $a.connect()$ updates action a's input and output dependencies, in case that changed inputs result in the action reading or modifying new objects. To ensure that past dependencies are not lost, *connect* only adds, and never removes, dependencies (even if the action in question does not use that dependency).

4.3 Refining actor objects: Finer-grained re-execution

An important goal of RETRO's design is minimizing re-execution, so as to avoid the need for user input to handle potential conflicts and external dependencies. It is often necessary to re-execute a subset of an actor's actions, but not necessarily the entire actor. For example, after rolling back a file like `/etc/passwd` to a checkpoint that was taken long ago, RETRO needs to replay all writes to that file, but should not need to re-execute the processes that issued those writes. Similarly, in Figure 2, RETRO would ideally re-execute only a part of `sshd` that checks whether Alice's password entry is the same, and if so, avoid re-executing the rest of `sshd`, which would lead to an external dependency because cryptographic keys would need to be re-negotiated. Unfortunately, re-executing a process from an intermediate state is difficult without process checkpointing.

To address this challenge, RETRO *refines* actors in the action history graph to explicitly denote parts of a process that can be independently re-executed. For example, RETRO models every system call issued by a process by a separate system call actor, comprising a single system call action, as shown in Figure 4. The system call arguments, and the result of the system call, are explicitly represented by system call argument and return value objects. This allows RETRO to re-execute individual system calls when necessary (e.g., to re-construct a file during repair), while avoiding re-execution of entire processes if the return values of system calls remain the same.

The same technique is also applied to re-execute specific functions instead of an entire process. Figure 5 shows a part of the action history graph for our running example,

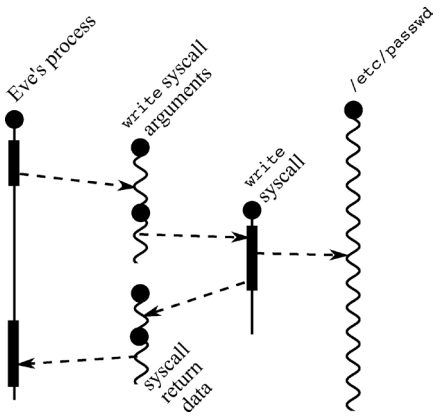


Figure 4: An illustration of the system call actor object and arguments and return value data objects, for Eve's write to the password file from Figure 2. Legend is the same as in Figure 2.

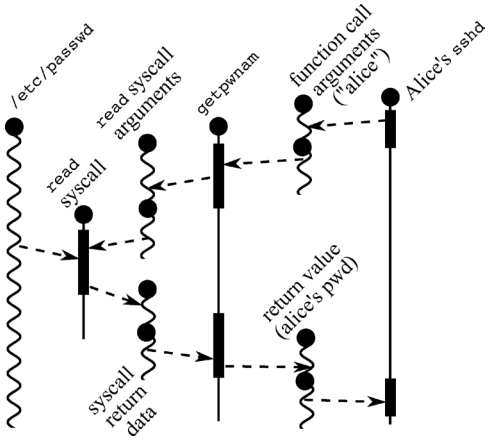


Figure 5: An illustration of refinement in an action history graph, depicting the use of additional actors to represent a re-executable call to `getpwnam` from `sshd`. Legend is the same as in Figure 2.

in which `sshd` creates a separate actor to represent its call to `getpwnam("alice")`. While `getpwnam`'s execution depends on the entire password file, and thus must be re-executed if the password file changes, its return value contains only Alice's password entry. If re-execution of `getpwnam` produces the same result, the rest of `sshd` need not be re-executed. §5 describes such higher-level managers in more detail.

The same mechanism helps RETRO create benign replacements for attack actions. For example, in order to undo a user accidentally executing malware, RETRO changes the `exec` system call's arguments to invoke `/bin/true` instead of the malware binary. To do this, RETRO synthesizes a new checkpoint for the object representing `exec`'s arguments, replacing the original malware binary path with `/bin/true`, and rolls back that object to the newly-created "checkpoint", as illustrated in Figure 6 and §4.5.

4.4 Refining data objects: Finer-grained data dependencies

While OS-level dependencies ensure completeness, they can be too coarse-grained, leading to false dependencies, such as every process depending on the `/tmp` directory. RETRO's design addresses this problem by *refining* the same state at different levels of abstraction in the graph when necessary. For instance, a directory manager creates individual objects for each file name in a directory, and helps disambiguate directory lookups and modifications by recording dependencies on specific file names.

The challenge in supporting refinement in the action history graph lies in dealing with multiple objects representing the same state. For example, the state of a single directory entry is a part of both the directory manager's object for that specific file name, as well as the file manager's node for that directory's inode. On one hand, we would like to avoid creating dependencies to and from the underlying directory inode, to prevent false dependencies. On the other hand, if some process does directly read the underlying directory inode's contents, it should depend on all of the directory entries in that directory.

To address this challenge, each object in RETRO keeps track of other objects that represent parts of its state. For example, the manager of each directory inode keeps track of all the directory entry objects for that directory. The object manager exposes this set of parts through the $o.parts$ property, as shown in Figure 3. In most cases, the manager tracks its parts through hierarchical names, as we discuss in §5.

RETRO's OS manager records all dependencies, even if the same dependency is also recorded by a higher-level manager. This means that RETRO can determine trust in higher-level dependencies at repair time. If the appropriate manager mediated all modifications to the larger object (such as a directory inode), and the manager was not compromised, RETRO can safely use finer-grained objects (such as individual directory entry objects). Otherwise, RETRO uses coarse-grained but safe OS-level dependencies.

4.5 Repair controller

RETRO uses a *repair controller* to repair system state with the help of object and action managers. Figure 6 summarizes the pseudo-code for the repair controller. The controller, starting from the REPAIR function, creates a parallel "repaired" timeline by re-executing actions in the order that they were originally executed. To do so, the controller maintains a set of objects that it is currently repairing (the *nodes* hash table), along with the last action that it performed on that object. REPAIRLOOP continuously attempts to re-execute the next action, until it has considered all actions, at which point the system state is fully repaired.

```
function ROLLBACK(node, checkpt)
    node.rollback(checkpt)
    state[node] := checkpt

function PREPAREREDO(action)
    if ¬action.connect() then return FALSE
    if state[action.actor] > action then
        cps := action.actor.checkpts
        cp := max(c ∈ cps | c ≤ action)
        ROLLBACK(action.actor, cp)
        return FALSE
    for all o ∈ (action.inputs ∪ action.outputs) do
        if state[o] ≤ action then continue
        ROLLBACK(o, max(c ∈ o.checkpts | c ≤ action))
        return FALSE
    return TRUE

function PICKACTION()
    actions := ∅
    for all o ∈ state | o is actor object do
        actions += min(a ∈ o.actions | a > state[o])
    for all o ∈ state | o is data object do
        actions += min(a ∈ o.readers ∪
                       o.writers | a > state[o])
    return min(actions)

function REPAIRLOOP()
    while a := PICKACTION() do
        if a.equiv() and state[o] ≥ a,
          ∀o ∈ a.outputs ∪ a.actor then
            for all i ∈ a.inputs ∩ keys(state) do
                state[i] := a
            continue   ▷ skip semantically-equivalent action
        if PREPAREREDO(a) then
            a.redo()
            for all o ∈ a.inputs ∪ a.outputs ∪ a.actor do
                state[o] := a

function REPAIR(repair_obj, repair_cp)
    ROLLBACK(repair_obj, repair_cp)
    REPAIRLOOP( )
```

Figure 6: The repair algorithm.

To choose the next action for re-execution, REPAIRLOOP invokes PICKACTION, which chooses the earliest action that hasn't been re-executed yet, out of all the objects being repaired. If the action's inputs are the same (according to *equiv*), and none of the outputs of the action need to be reconstructed, REPAIRLOOP does not re-execute the action, and just advances the state of the action's input nodes. If the action needs to be re-executed, REPAIRLOOP invokes PREPAREREDO, which ensures that the action's actor, input objects, and output objects are all in the right state to re-execute the action (by rolling back these objects when appropriate). Once PREPAREREDO indicates it is ready, REPAIRLOOP re-executes the action and updates the state of the actor, input, and output

objects. Finally, REPAIR invokes REPAIRLOOP in the first place, after rolling back $repair\_obj$ to the (newly-synthesized) checkpoint $repair\_cp$, as described in §4.3.

Not shown in the pseudo-code is handling of refined objects. When the controller rolls back an object that has a non-empty set of parts, it must consider re-executing actions associated with those parts, in addition to actions associated with the larger object. Also not shown is the checking of integrity for higher-level dependencies, as described in §4.4.

5 OBJECT AND ACTION MANAGERS

This section describes RETRO's object and action managers, starting with the file system and OS managers that guarantee completeness of the graph, and followed by higher-level managers that provide finer-grained dependencies for application-specific parts of the graph.

5.1 File system manager

The file system manager is responsible for all file objects. To uniquely identify files, the manager names file objects by $\langle device, part, inode \rangle$. The *device* and *part* components identify the disk and partition holding the file system. Our current prototype disallows direct access to partition block devices, so that file system dependencies are always trusted. The *inode* number identifies a specific file by inode, without regard to path name. To ensure that files can be uniquely identified by inode number, the file system manager prevents inode reuse until all checkpoints and logs referring to the inode have been garbage-collected.

During normal operation, the file system manager must periodically checkpoint its objects (including files and directories), using any checkpointing strategy. Our implementation relies on a snapshotting file system to make periodic snapshots of the entire file system tree (e.g., once per day). This works well for systems which already create daily snapshots [26, 32, 37, 38], where the file system manager can simply leverage existing snapshots. Upon file deletion, the file system manager moves the deleted inode into a special directory, so that it can reuse the same exact inode number on rollback. The manager preserves the inode's data contents, so that RETRO can undo an unlink operation by simply linking the inode back into a directory (see §5.3).

During repair, the file system manager's $rollback$ method uses a special kernel module to open the checkpointed file as well as the current file by their inode number. Once the repair manager obtain a file descriptor for both inodes, it overwrites the current file's contents with the checkpoint's contents, or re-constructs an identical set of directory entries, for directory inodes. On rollback to a file system snapshot where the inode in question was not allocated yet, the file system manager truncates the file to zero bytes, as if it was freshly created. As a precaution, the file system manager creates a new file system snapshot before initiating any rollback.

5.2 OS manager

The OS manager is responsible for process and system call actors, and their actions. The manager names each process in the graph by $\langle bootgen, pid, pidgen, execgen \rangle$. *bootgen* is a boot-up generation number to distinguish process IDs across reboots. *pid* is the Unix process ID, and *pidgen* is a generation number for the process ID, used to distinguish recycled process IDs. Finally, *execgen* counts the number of times a process called the `exec` system call; the OS manager logically treats `exec` as creating a new process, albeit with the same process ID. The manager names system calls by $\langle bootgen, pid, pidgen, execgen, sysid \rangle$, where *sysid* is a per-process unique ID for that system call invocation.

5.2.1 Recording normal execution

During normal execution, the OS manager intercepts and records all system calls that create dependencies to or from other objects (i.e., not `getpid`, etc), recording enough information about the system calls to both re-execute them at repair time, and to check whether the inputs to the system call are semantically equivalent. The OS manager creates nominal checkpoints of process and system call actors. Since checkpointing of processes mid-execution is difficult [13, 35], our OS manager checkpoints actors only in their "initial" state immediately prior to `exec`, denoted by $\perp$. The OS manager also keeps track of objects representing ephemeral state, including pipes and special devices such as `/dev/null`. Although RETRO does not attempt to repair this state, having these objects in the graph helps track and check dependencies using $equiv$ during repair, and to perform partial re-execution.

5.2.2 Action history graph representation

In the action history graph, the OS manager represents each system call by two actions in the process actor, two intermediate data objects, and a system call actor and action, as shown in Figure 4. The first process action, called the *syscall invocation* action, represents the execution of the process up until it invokes the system call. This action conceptually places the system call arguments, and any other relevant state, into the system call arguments object. For example, the arguments for a file write include the target inode, the offset, and the data. The arguments for `exec`, on the other hand, include additional information that allows re-executing the system call actor without having to re-execute the process actor, such as the current working directory, file descriptors not marked `O_CLOEXEC`, and so on.

The system call action, in a separate actor, conceptually reads the arguments from this object, performs the system call (incurring dependencies to corresponding objects), and writes the return value and any returned data into the return value object. For example, a write system call action, shown in Figure 4, creates a dependency to the modified file, and stores the number of bytes written into the return value object. Finally, the second process action, called the *syscall return* action, reads the returned data from that object, and resumes process execution. In case of `fork` or `exec`, the OS manager creates two return objects and two syscall return actions, representing return values to both the old and new process actors. Thus, every process actor starts with a syscall return action, with a dependency from the return object for `fork` or `exec`.

In addition to system calls, Unix processes interact with memory-mapped files. RETRO cannot re-execute memory-mapped file accesses without re-executing the process. Thus, the OS manager associates dependencies to and from memory-mapped files with the process's own actions, as opposed to actions in a system call actor. In particular, every process action (either syscall invocation or return) has a dependency *from* every file memory-mapped by the process at that time, and a dependency *to* every file memory-mapped as writable at that time.

5.2.3 Shepherded re-execution

During repair, the OS manager must re-execute two types of actors: process actors and system call actors. For system call actors, when the repair controller invokes $redo$, the OS manager reads the (possibly changed) values from the system call arguments object, executes the system call in question, and places return data into the return object. $equiv$ on a system call action checks whether the input objects have the same values as during the original execution. Finally, $connect$ reads the (possibly changed) inputs, and creates any new dependencies that result. For example, if a `stat` system call could not find the named file during original execution, but RETRO restores the file during repair, $connect$ would create a new dependency from the newly-restored file.

For process actors, the OS manager represents the state of a process during repair with an actual process being *shepherded* via the `ptrace` debug interface. On $p.rollback(\perp)$, the OS manager creates a fresh process for process object p under `ptrace`. When the repair controller invokes $redo$ on a syscall return action, the OS manager reads the return data from the corresponding system call return object, updates the process state using `PTRACE_POKEDATA` and `PTRACE_SETREGS`, and allows the process to execute until it's about to invoke the next system call. $equiv$ on a system call return action checks if the data in the system call return object is the same as during the original execution. When the repair controller invokes $redo$ on the subsequent syscall invocation action, the OS manager simply marshals the arguments for the system call invocation into the corresponding system call arguments object. This allows the repair controller to separately schedule the re-execution of the system call, or to re-use previously recorded return data. Finally, $connect$ does nothing for process actions.

One challenge for the OS manager is to deal with processes that issue different system calls during re-execution. The challenge lies in matching up system calls recorded during original execution with system calls actually issued by the process during re-execution. The OS manager employs greedy heuristics to match up the two system call streams. If a new syscall does not match a previously-recorded syscall in order, the OS manager creates new system call actions, actors, and objects (as shown in Figure 4). Similarly, if a previously-recorded syscall does not match the re-executed system calls in order, the OS manager replaces the previously-recorded syscall's actions with no-ops. In the worst case, the only matches will be the initial return from `fork` or `exec`, and the final syscall invocation that terminates the process, potentially leading to more re-execution, but not a loss of correctness.

In our running example, Eve trojans the `texi2pdf` shell script by adding an extra line to start her botnet worker. After repairing the `texi2pdf` file, RETRO re-executes every process that ran the trojaned `texi2pdf`. During shepherded re-execution of `texi2pdf`, `exec` system calls to legitimate LaTeX programs are identical to those during the original execution; in other words, the system call argument objects are equivalent, and $equiv$ on the system call action returns true. As a result, there is no need to re-execute these child processes. However, `exec` system calls to Eve's bot are missing, so the manager replaces them with no-ops, which recursively undoes any changes made by Eve's bot.

5.3 Directory manager

The directory manager is responsible for exposing finer-grained dependency information about directory entries. Although the file system manager tracks changes to directories, it treats the entire directory as one inode, causing false dependencies in shared directories like `/tmp`. The directory manager names each directory entry by $\langle device, part, inode, name \rangle$. The first three components of the name are the file system manager's name for the directory inode. The *name* part represents the file name of the directory entry.

During normal operation, the directory manager must record checkpoints of its objects, conceptually consisting of the inode number for the directory entry (or $\perp$ to represent non-existent directory entries). However, since the file system manager already records checkpoints of all directories, the directory manager relies on the file

system manager's checkpoints, and does not perform any checkpointing of its own. The directory manager similarly relies on the OS manager to record dependencies between system call actions and directory entries accessed by those system calls, such as name lookups in `namei` (which incur a dependency from every directory entry traversed), or directory modifications by `rename` (which incur a dependency to the modified directory entries).

During repair, the directory manager's sole responsibility is rolling back directory entries to a checkpoint; the OS manager handles redo of all system calls. To roll back a directory entry to an earlier checkpoint, the directory manager finds the inode number contained in that directory entry (using the file system manager's checkpoint), and changes the directory entry in question to point to that inode, with the help of RETRO's kernel module. If the directory entry did not exist in the checkpoint, the directory manager similarly unlinks the directory entry.

5.4 System library managers

Every user login on a typical Unix system accesses several system-wide files. For example, each login attempt accesses the entire password file, and successful logins update both the `utmp` file (tracking currently logged in users) and the `lastlog` file (tracking each user's last login). In a naïve system, these shared files can lead to false dependencies, making it difficult to disambiguate attacker actions from legitimate changes. To address this problem, RETRO uses a *libc* system library manager to expose the semantic independence between these actions.

One strawman approach would be to represent such shared files much as directories (i.e., creating a separate object for each user's password file entry). However, unlike the directory manager, which mediates all accesses to a directory, a manager for a function in *libc* cannot guarantee that an attacker will not bypass it—the manager, libc, and the attacker can be in the same address space. Thus, the `libc` manager does not change the representation of data objects, and instead simplifies re-execution, by creating actors to represent the execution of individual *libc* functions. For example, Figure 5 shows an actor for the `getpwnam` function call as part of `sshd`.

During normal operation, the library manager creates a fresh actor for each function call to one of the managed functions, such as `getpwnam`, `getspnam`, and `getgrouplist`. The library manager names function call actors by $\langle bootgen, pid, pidgen, execgen, callgen \rangle$; the first four parts name the process, and *callgen* is a unique ID for each function call. Much as with system call actors, the arguments object contains the function name and arguments, and the return object contains the return value. Like processes, function call actors have only one checkpoint, $\perp$, representing their initial state prior to the call.

The library manager requires the OS manager's help to associate system calls issued from inside library functions with the function call actor, instead of the process actor. To do this, the OS manager maintains a "call stack" of function call actors that are currently executing. On every function call, the library manager pushes the new function call actor onto the call stack, and on return, it pops the call stack. The OS manager associates syscall invocation and return actions with the last actor on the call stack, if any, instead of the process actor.

During repair, the library manager's $rollback$ and $redo$ methods allow the repair controller to re-execute individual functions. For example, in Figure 5, the controller will re-execute `getpwnam`, because its dependency on `/etc/passwd` changed due to repair. However, if $equiv$ indicates the return value from `getpwnam` did not change, the controller need not re-execute the rest of `sshd`.

RETRO's trust assumption about the library manager is that the function does not semantically affect the rest of the program's execution other than through its return value. If an attacker process compromises its own *libc* manager, this does not pose a problem, because the process already depended on the attacker in other ways, and RETRO will repair it. However, if an attacker exploits a vulnerability in the function's input parsing code (such as a buffer overflow in `getpwnam` parsing `/etc/passwd`), it can take control of `getpwnam`, and influence the execution of the process in ways other than `getpwnam`'s return value. Thus, RETRO trusts *libc* functions wrapped by the library manager to safely parse files and faithfully represent their return values.

5.5 Terminal manager

Undoing attacker's actions during repair can result in legitimate applications sending different output to a user's terminal. For example, if the user ran `ls /tmp`, the output may have included temporary files created by the attacker, or the `ls` binary was trojaned by the attacker to hide certain files. While RETRO cannot undo what the user already saw, the terminal manager helps RETRO generate compensating actions.

The terminal manager is responsible for objects representing pseudo-terminal, or pty, devices (`/dev/pts/`*N* in Linux). During normal operation, the manager records the user associated with each pty (with help from `sshd`), and all output sent to the pty. During repair, if the output sent to the pty differs from the output recorded during normal operation, the terminal manager computes a text diff between the two outputs, and emails it to the user.

5.6 Network manager

The network manager is responsible for compensating for externally-visible changes. To this end, the network manager maintains objects representing the outside world (one object for each TCP connection, and one object for

each IP address/UDP port pair). During normal operation, the network manager records all traffic, similar to the terminal manager.

During repair, the network manager compares repaired outgoing data with the original execution. When the network manager detects a change in outgoing traffic, it flags an external dependency, and presents the user or administrator with three choices. The first choice is to ignore the dependency, which is appropriate for network connections associated with the adversary (such as Eve's login session in our running example, which will generate different network traffic during repair). The second choice is to re-send the network traffic, and wait for a response from the outside world. This is appropriate for outgoing network connections and idempotent protocols, such as DNS. Finally, the third choice is to require the user to manually resolve the external dependency, such as by manually re-playing the traffic for incoming connections. This is necessary if, say, the response to an incoming SMTP connection has changed, the application did not provide its own compensating action, and the user does not want to ignore this dependency.

6 IMPLEMENTATION

We implemented a prototype of RETRO for Linux,[3] components of which are summarized in Figure 7. During normal execution, a kernel module intercepts and records all system calls to a log file, implementing the runtime half of the OS, file system, directory, terminal, and network managers. To allow incremental loading of log records, RETRO records an index alongside the log file that allows efficient lookup of records for a given process ID or inode number. The file system manager implements checkpoints using subvolume snapshots in btrfs [37]. The libc manager logs function calls using a new RETRO system call to add ordered records to the system-wide log. The repair controller, and the repair-time half of each manager, are implemented as Python modules.

RETRO implements three optimizations to reduce logging costs. First, it records SHA-1 hashes of data read from files, instead of the actual data. This allows checking for equivalence at repair time, but avoids storing the data twice. Second, it does not record data read or written by white-listed deterministic processes (in our prototype, this includes `gcc` and `ld`). This means that, if any of the read or write dependencies to or from these processes are suspected during repair, the entire process will have to be re-executed, because individual read and write system calls cannot be checked for equivalence or re-executed. Since all of the dependency relationships are preserved, this optimization trades off repair time for recording time, but does not compromise completeness. Third, RETRO compresses the resulting log files to save space.

[3] While our prototype is Linux-specific, we believe that RETRO's approach is equally applicable to other operating systems.

Component	Lines of code
Logging kernel module	3,300 lines of C
Repair controller, manager modules	5,000 lines of Python
System library managers	700 lines of C
Backtracking GUI tool	500 lines of Python

Figure 7: Components of our RETRO prototype, and an estimate of their complexity, in terms of lines of code.

Attack	Objects repaired with predicates			Objects repaired without predicates			User input
	Proc	Func	File	Proc	Func	File	
Password change	1	2	4	430	20	274	1
Log cleaning	59	0	40	60	0	40	0
Running example	58	57	75	513	61	300	1
`sshd` trojan	530	47	303	530	47	303	3

Figure 8: Repair statistics for the two honeypot attacks (top) and two synthetic attacks (bottom). The repaired objects are broken down into processes, functions (from libc), and files. Intermediate objects such as syscall arguments are not shown. The concurrent workload consisted of 1,261 process, function, and file objects (both actor and data objects), and 16,239 system call actions. RETRO was able to fully repair all attacks, with no false positives or false negatives. User input indicate the number of times RETRO asked for user assistance in repair; the nature of the conflict is reported in §7.

7 EVALUATION

This section answers three questions about RETRO, in turn. First, what kinds of attacks can RETRO recover from, and how much user input does it require? Second, are all of RETRO's mechanisms necessary in practice? And finally, what are the performance costs of RETRO, both during normal execution and during repair?

7.1 Recovery from attack

To evaluate how RETRO recovers from different attacks, we used three classes of attack scenarios. First, to make sure we can repair real-world attacks, we used attacks recorded by a honeypot. Second, to make sure RETRO can repair worst-case attacks, we used synthetic attacks designed to be particularly challenging for RETRO, including the attack from our running example. For both real-world and synthetic attacks, we perform user activity described in the running example after the attack takes place—namely, root logs in via ssh and adds an account for Alice, who then also logs in via ssh to edit and build a LaTeX file. Finally, we compare RETRO to Taser, the state-of-the-art attack recovery system, using attack scenarios from the Taser paper [17].

Honeypot attacks. To collect real-world attacks, we ran a honeypot [1] for three weeks, with a modified `sshd` that accepted any password for login as root. Out of many root logins, we chose two attacks that corrupted our honeypot's state in the most interesting ways.[4] In the first attack, the attacker changed the root password. In the second attack, the attacker downloaded and ran a Linux

[4] Most of the attackers simply ran a botnet binary or a port scanner.

Scenario	Taser				RETRO	User input required
	Snapshot	NoI	NoIAN	NoIANC		
Illegal storage	FP	FP	FN	FN	✓	None.
Content destruction	FP	✓	✓	FN	✓	None. (Generates terminal diff compensating action.)
Unhappy student	FP	FP	✓	FN	✓	None. (Generates terminal diff compensating action.)
Compromised database	FP	FP	FP	FN	✓	None.
Software installation	FP	FP	✓	✓	✓	Re-execute browser (or ignore browser state changes).
Inexperienced admin	FP	FP	FP	✓	✓	Skip re-execution of attacker's login session.

Figure 9: A comparison of Taser's four policies and RETRO against a set of scenarios used to evaluate Taser [17]. Taser's snapshot policy tracks all dependencies, NoI ignores IPC and signals, NoIAN also ignores file name and attributes, and NoIANC further ignores file content. FP indicates a false positive (undoing legitimate actions), FN indicates a false negative (missing parts of the attack), and ✓ indicates no false positives or negatives.

binary that scrubbed system log files of any mention of the attacker's login attempt.

For both of these attacks, RETRO was able to repair the system while preserving all legitimate user actions, as summarized in Figure 8. In the password change attack, root was unable to log in after the attack, immediately exposing the compromise, although we still logged in as Alice and ran `texi2pdf`. In the second attack, all 59 repaired processes were from the attacker's log cleaning program, whose effects were undone.

For these real-world attacks, RETRO required minimal user input. RETRO required one piece of user input to repair the password change attack, because root's login attempt truly depended on root's entry in `/etc/passwd`, which was modified by the attacker. In our experiment, the user told the network manager to ignore the conflict. RETRO required no user input for the log cleaning attack.

Synthetic attacks. To check if RETRO can recover from more insidious attacks, we constructed two synthetic attacks involving trojans; results for both are summarized in Figure 8. For the first synthetic attack, we used the running example, where the attacker adds an account for `eve`, installs a botnet and a backdoor PHP script, and trojans the `/usr/bin/texi2pdf` shell script to restart the botnet. Legitimate users were unaware of this attack, and performed the same actions. Once the administrator detected the attack, RETRO reverted Eve's changes, including the `eve` account, the bot, and the trojan. As described in §5.2.3, RETRO used shepherded re-execution to undo the effects of the trojan without re-running the bulk of the trojaned application. As Figure 8 indicates, RETRO re-executed several functions (`getpwnam`) to check if removing `eve`'s account affected any subsequent logins. One login session was affected—Eve's login—and RETRO's network manager required user input to confirm that Eve's login need not be re-executed.

One problem we discovered when repairing the running example attack is that the UID chosen for Alice by root's `useradd alice` command depends on whether `eve`'s account is present. If RETRO simply re-executed `useradd alice`, `useradd` would pick a different UID during re-execution, requiring RETRO to re-execute Alice's entire session. Instead, we made the `useradd` command part of the system library manager, so that during repair, it first tries to re-execute the action of adding user `alice` under the original UID, and only if that fails does it re-execute the full `useradd` program. This ensures that Alice's UID remains the same even after RETRO removes the `eve` account (as long as Alice's UID is still available).

A second synthetic attack we tried was to trojan `/usr/sbin/sshd`. In this case, users were able to log in as usual, but undoing the attack required re-executing their login sessions with a good `sshd` binary. Because RETRO cannot rerun the remote ssh clients (and a new key exchange, resulting in different keys, makes TCP-level replay useless), RETRO's network manager asks the administrator to redo each ssh session manually. Of course, this would not be practical on a real system, and the administrator may instead resort to manually auditing the files affected by those login sessions, to verify whether they were affected by the attack in any way. However, we believe it is valuable for RETRO to identify all connections affected by the attack, so as to help the administrator locate potentially affected files. In practice, we hope that an intrusion detection system can notice such wide-reaching attacks; after a few user logins, the dependency graph indicates that unrelated user logins are all dependent on a previous login session, which an IDS may be able to flag.

Taser attacks. Finally, we compare RETRO to the state-of-the-art intrusion recovery system, Taser, under the attack scenarios that were used to originally evaluate Taser [17]. Figure 9 summarizes the results.

In the first scenario, illegal storage, the attacker creates a new account for herself, stores illegal content on the system, and trojans the `ls` binary to mask the illegal content. RETRO rolls back the account, illegal files, and the trojaned `ls` binary, and uses the legitimate `ls` binary to re-execute all `ls` processes from the past. Even though the trojaned `ls` binary hid some files, the legitimate `ls` binary produces the same output, because RETRO removes the hidden files during repair. As a result, there is no need to notify the user. If `ls`'s output did change, the terminal manager would have sent a diff to the affected users.

In the content destruction scenario, an attacker deletes a user's files. Once the user notices the problem, he uses RETRO to undo the attack. After recovering the

Workload	Without RETRO	With RETRO		Log size	Snapshot size	# of objects	# of actions
	1 core	1 core	2 cores				
Kernel build	295 sec	557 sec	351 sec	761 MB	308 MB	87,405	5,698,750
Web server	7260 req/s	3195 req/s	5453 req/s	98 MB	272 KB	508	185,315
HotCRP	20.4 req/s	15.1 req/s	20.0 req/s	81 MB	27 MB	19,969	939,418

Figure 10: Performance and storage costs of RETRO for three workloads: building the Linux kernel, serving files as fast as possible using Apache [2] for 1 minute, and simulating requests to HotCRP [23] from the 30 minutes before the SOSP 2007 deadline, which averaged 2.1 requests per second [44] (running as fast as possible, this workload finished in 3–4 minutes). "# of objects" reflects the number of files, directory entries, and processes; not included are intermediate objects such as system call arguments. "# of actions" reflects the number of system call actions.

files, RETRO generates a terminal output diff for the login session during which the user noticed the missing files (after repair, the user's `ls` command displays those files).

In the unhappy student scenario, a student exploits an `ftpd` bug to change permissions on a professor's grade file, then modifies the grade file in another login session, and finally a second accomplice user logs in and makes a copy of the grade file. In repairing the attack, RETRO rolls back the grade file and its permissions, re-executes the copy command (which now fails), and uses the terminal manager to generate a diff for the attackers' sessions, informing them that their copy command now failed.

In the compromised database scenario, an attacker breaks into a server, modifies some database records (in our case we used SQLite), and subsequently a legitimate user logs in and runs a script that updates database records of its own. RETRO rolls back the database file to a state before the attack, and re-executes the database update script to preserve subsequent changes, with no user input.

In the software installation scenario, the administrator installs the wrong browser plugin, and only detects this problem after running the browser and downloading some files. During repair, RETRO rolls back the incorrect plugin, and attempts to repair the browser using re-execution. Since RETRO encounters external dependencies in re-executing network applications, it requests the user to manually redo any interactions with the browser. In our experiment, the user ignored this external dependency, because he knew the browser made no changes to local state worth preserving.

In the inexperienced admin scenario, root selects a weak password for a user account, and an attacker guesses the password and logs in as the user. Undoing root's password change affects the attacker's login session, requiring one user input to confirm to the network manager that it's safe to discard the attacker's TCP connection.

In summary, RETRO correctly repairs all six attack scenarios posed by Taser, requiring user input only in two cases: to re-execute the browser, and to confirm that it's safe to drop the attacker's login session. Taser requires application-specific policies to repair these attacks, and some attacks cannot be fully repaired under any policy. Taser's policies also open up the system to false negatives, allowing an adversary to bypass Taser altogether.

7.2 Technique effectiveness

In this subsection, we evaluate the effectiveness of RETRO's specific techniques, including re-execution, predicate checking, and refinement.

Re-execution is key to preserving legitimate user actions. As described in §7.1 and quantified in Figure 8, RETRO re-executes several processes and functions to preserve and repair legitimate changes. Without re-execution, RETRO would have to conservatively roll back any files touched by the process in question, much like Taser's snapshot policy, which incurs false positives.

Without predicates, RETRO would have to perform conservative dependency propagation in the dependency graph. As in Taser, dependencies on attack actions quickly propagate to most objects in the graph, requiring re-execution of almost every process. This leads to re-execution of `sshd`, which requires user assistance. Figure 8 shows that many of the objects repaired without predicates were not repaired with predicates enabled. Taser would roll back all of these objects (false positives). Thus, predicates are an important technique to minimize user input due to re-execution.

Without refinement of actor and data objects, RETRO would incur false dependencies via `/tmp` and `/etc/passwd`. As Figure 8 shows, several functions (such as `getpwnam`) were re-executed in repairing from attacks. If RETRO was unable to re-execute just those functions, it would have re-executed processes like `sshd`, forcing the network manager to request user input. Thus, refinement is important to minimizing user input due to false dependencies.

7.3 Performance

We evaluate RETRO's performance costs in two ways. First, we consider costs of RETRO's logging during normal execution. To this end, we measure the CPU overhead and log size for several workloads. Figure 10 summarizes the results. We ran our experiments on a 2.8GHz Intel Core i7 system with 8 GB RAM running a 64-bit Linux 2.6.35 kernel, with either one or two cores enabled.

The worst-case workload for RETRO is a system that uses 100% of CPU time and spends most of its time communicating between small processes. One such extreme workload is a system that continuously re-builds the Linux kernel; another example is an Apache server continuously

serving small static files. For such systems, RETRO incurs a 89–127% CPU overhead using a single core, and generates about 100–150 GB of logs per day. A 2 TB disk ($100) can store two weeks of logs at this rate before having to garbage-collect older log entries. If a spare second core is available, and the application cannot take advantage of it, it can be used for logging, resulting in only 18–33% CPU overhead.

For a more realistic application, such as a HotCRP [23] paper submission web site, RETRO incurs much less overhead, since HotCRP's PHP code is relatively CPU-intensive. If we extrapolate the workload from the 30 minutes before the SOSP 2007 deadline [44] to an entire day, HotCRP would incur 35% CPU overhead on a single core (and almost no overhead if an additional unused core were available), and use about 4 GB of log space per day. We believe that these are reasonable costs to pay to be able to recover integrity after a compromise of a paper submission web site.

Second, we consider the time cost of repairing a system using RETRO after an attack. As Figure 8 illustrated, RETRO is often effective at repairing only a small subset of objects and actions in the action history graph, and for attacks that affect the entire system state, such as the `sshd` trojan, user input dominates repair costs. To illustrate the costs of repairing a subset of the action history graph, we measure the time taken by RETRO to repair from a micro-benchmark attack, where the adversary adds an extraneous line to a log file, which is subsequently modified by a legitimate process. When only this attack is present in RETRO's log (consisting of 10 process objects, 126 file objects, and 399 system call actions), repair takes 0.3 seconds. When this attack runs concurrently with a kernel build (as shown in Figure 10), repair of the attack takes 4.7 seconds (10× longer), despite the fact that the log is 10,000× larger. This shows that RETRO's log indexing makes repair time depend largely on the number of affected objects, rather than the overall log size.

8 DISCUSSION AND FUTURE WORK

An important assumption of RETRO is that the attacker does not compromise the kernel. Unfortunately, security vulnerabilities are periodically discovered in the Linux kernel [5, 6], making this assumption potentially dangerous. One solution may be to use virtual machine based techniques [14, 21], although it is difficult to distinguish kernel objects after a kernel compromise. We plan to explore ways of reducing trust in future work.

In our current prototype, if attackers compromise the kernel and obtain access to RETRO's log files, they may be able to extract sensitive information, such as user passwords or keys, that would not have been persistently stored on a system without RETRO. One possible solution may be to encrypt the log files and checkpoints, so that the administrator must reboot the system from a trusted CD and enter the password to initiate recovery.

Our current prototype can only repair the effects of an attack on a single machine, and relies on compensating actions to repair external state. In future work, we plan to explore ways to extend automated repair to distributed systems, perhaps based on the ideas from [29, 42].

RETRO requires the system administrator to specify the initial intrusion point in order to undo the effects of the attack, and finding the initial intrusion point can be difficult. In future work, we hope to leverage the extensive data available in RETRO's dependency graph to build intrusion detection tools that can better pin-point intrusions. Alternatively, instead of trying to pinpoint the attack, we may be able to use RETRO to retroactively apply security patches into the past, and re-execute any affected computations, thus eliminating any attacks that exploited the vulnerability in question.

We did not have space to address several practical aspects of using RETRO, such as performing multiple repairs or undoing a repair. These operations translate into making additional checkpoints, and updating the graph accordingly after repair. Also, as hinted at in §5, we plan to explore the use of more specialized repair managers, such as managers for a language runtime, a database, or an application like a web server or web browser. Finally, while RETRO's performance and storage overheads are already acceptable for some workloads, we plan to further reduce them by not logging intermediate dependencies that can be reconstructed at repair time.

9 CONCLUSION

RETRO repairs system integrity from past attacks by using an action history graph to track system-wide dependencies, roll back affected objects, and re-execute legitimate actions affected by the attack. RETRO minimizes user input by avoiding re-execution whenever possible, and by using compensating actions for external dependencies. RETRO's key techniques for minimizing re-execution include predicates, refinement, and shepherded re-execution. A prototype of RETRO for Linux recovers from a mix of ten real-world and synthetic attacks, repairing all side-effects of the attack in all cases. Six attacks required no user input to repair, and RETRO required significant user input in only two cases involving trojaned network-facing applications.

ACKNOWLEDGMENTS

We thank Victor Costan, Robert Morris, Jacob Strauss, the anonymous reviewers, and our shepherd, Adrian Perrig, for their feedback. Quanta Computer partially supported this work. Taesoo Kim is partially supported by the Samsung Scholarship Foundation, and Nickolai Zeldovich is partially supported by a Sloan Fellowship.

REFERENCES

[1] The Honeynet Project. http://www.honeynet.org/.

[2] Apache web server, May 2010. http://httpd.apache.org/.

[3] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1167–1185, 2002.

[4] Apple Inc. What is Mac OS X - Time Machine. http://www.apple.com/macosx/what-is-macosx/time-machine.html.

[5] J. Arnold and M. F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proc. of the ACM EuroSys Conference*, Nuremberg, Germany, Mar 2009.

[6] J. Arnold, T. Abbott, W. Daher, G. Price, N. Elhage, G. Thomas, and A. Kaseorg. Security impact ratings considered harmful. In *Proc. of the 12th Workshop on Hot Topics in Operating Systems*, Monte Verita, Switzerland, May 2009.

[7] AVG Technologies. Why traditional anti-malware solutions are no longer enough. http://download.avg.com/filedir/other/pf_wp-90_A4_us_z3162_20091112.pdf, Oct 2009.

[8] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corp., Bedford, MA, Apr 1977.

[9] U. Braun, A. Shinnar, and M. Seltzer. Securing provenance. In *Proc. of the 3rd Usenix Workshop on Hot Topics in Security*, San Jose, CA, Jul 2008.

[10] A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proc. of the 2003 Usenix ATC*, pages 1–14, San Antonio, TX, Jun 2003.

[11] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. Lam. The Collective: A cache-based system management architecture. In *Proc. of the 2nd NSDI*, pages 259–272, Boston, MA, May 2005.

[12] CheckPoint, Inc. IPS-1 intrusion detection and prevention system. http://www.checkpoint.com/products/ips-1/.

[13] J. Corbet. A checkpoint/restart update. http://lwn.net/Articles/375855/, Feb 2010.

[14] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. of the 5th OSDI*, pages 211–224, Boston, MA, Dec 2002.

[15] S. Forrest, S. Hofmeyr, and A. Somayaji. The evolution of system-call monitoring. In *Proc. of the 2008 Annual Computer Security Applications Conference*, pages 418–430, Dec 2008.

[16] FreeBSD. What is securelevel? http://www.freebsd.org/doc/en_US.ISO8859-1/books/faq/security.html#SECURELEVEL.

[17] A. Goel, K. Po, K. Farhadi, Z. Li, and E. D. Lara. The Taser intrusion recovery system. In *Proc. of the 20th ACM SOSP*, pages 163–176, Brighton, UK, Oct 2005.

[18] B. Harder. Microsoft Windows XP system restore. http://msdn.microsoft.com/en-us/library/ms997627.aspx, Apr 2001.

[19] A. Joshi, S. King, G. Dunlap, and P. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proc. of the 20th ACM SOSP*, pages 91–104, Brighton, UK, Oct 2005.

[20] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proc. of the 2nd ACM CCS*, pages 18–29, Fairfax, VA, Nov 1994.

[21] S. T. King and P. M. Chen. Backtracking intrusions. *ACM TOCS*, 23(1):51–76, Feb 2005.

[22] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In *Proc. of the 12th NDSS*, San Diego, CA, Feb 2005.

[23] E. Kohler. Hot crap! In *Proc. of the Workshop on Organizing Workshops, Conferences, and Symposia for Computer Systems*, San Francisco, CA, Apr 2008.

[24] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proc. of the 18th Usenix Security Symposium*, Montreal, Canada, Aug 2009.

[25] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. of the 21st ACM SOSP*, pages 321–334, Stevenson, WA, Oct 2007.

[26] A. Lewis. LVM HOWTO: Snapshots. http://www.tldp.org/HOWTO/LVM-HOWTO/snapshotintro.html.

[27] P. Liu, P. Ammann, and S. Jajodia. Rewriting histories: Recovering from malicious transactions. *Journal of Distributed and Parallel Databases*, 8(1):7–40, 2000.

[28] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. of the 2001 Usenix ATC*, pages 29–40, Jun 2001. Freenix track.

[29] P. Mahajan, R. Kotla, C. C. Marshall, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber. Effective and efficient compromise recovery for weakly consistent replication. In *Proc. of the ACM EuroSys Conference*, pages 131–144, Nuremberg, Germany, Mar 2009.

[30] Microsoft. How to use the roll back driver feature in Windows XP. http://support.microsoft.com/kb/283657, Aug 2007.

[31] MokaFive, Inc. Mokafive, virtual desktops for businesses and personal use. http://www.mokafive.com/.

[32] NetApp. Snapshot. http://www.netapp.com/us/products/platform-os/snapshot.html.

[33] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proc. of the 20th ACM SOSP*, Brighton, UK, Oct 2005.

[34] R. Paleari, L. Martignoni, E. Passerini, D. Davidson, M. Fredrikson, J. Giffin, and S. Jha. Automatic generation of remediation procedures for malware infections. In *Proc. of the 19th Usenix Security Symposium*, Washington, DC, Aug 2010.

[35] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Proc. of the 1995 Usenix ATC*, pages 213–223, New Orleans, LA, Jan. 1995.

[36] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. In *Proc. of the 22nd ACM SOSP*, pages 161–176, Big Sky, MT, Oct 2009.

[37] O. Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage*, 3(4):1–27, 2008.

[38] M. Satyanarayanan. Scalable, secure and highly available file access in a distributed workstation environment. *IEEE Computer*, pages 9–21, May 1990.

[39] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. of the 21st ACM SOSP*, Stevenson, WA, Oct 2007.

[40] F. Shafique, K. Po, and A. Goel. Correlating multi-session attacks via replay. In *Proc. of the Second Workshop on Hot Topics in System Dependability*, Seattle, WA, Nov 2006.

[41] B. Spengler. grsecurity. http://www.grsecurity.net/.

[42] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proc. of the 14th NDSS*, San Diego, CA, Feb-Mar 2007.

[43] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proc. of the 14th ACM CCS*, Alexandria, VA, Oct-Nov 2007.

[44] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proc. of the 22nd ACM SOSP*, pages 291–304, Big Sky, MT, Oct 2009.

[45] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th OSDI*, pages 263–278, Seattle, WA, Nov 2006.

Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications

Adam Chlipala
Impredicative LLC

Abstract

We present a system for sound static checking of security policies for database-backed Web applications. Our tool checks a combination of access control and information flow policies, where the policies vary based on database contents. For instance, one or more database tables may represent an access control matrix, controlling who may read or write which cells of these and other tables. Using symbolic evaluation and automated theorem-proving, our tool checks these policies statically, requiring no program annotations (beyond the policies themselves) and adding no run-time overhead. Specifications come in the form of *SQL queries as policies*: for instance, an application's confidentiality policy is a fixed set of queries, whose results provide an upper bound on what information may be released to the user. To provide user-dependent policies, we allow queries to depend on *what secrets the user knows*. We have used our prototype implementation to check several programs representative of the data-centric Web applications that are common today.

1 Introduction

Much of today's most important software exists as Web applications, and many of these applications are thin interface layers for relational databases. Real-world requirements impel developers to implement many application-specific schemes for access control ("who can do what?") and information flow ("who can learn what?"). To reason about correctness of these implementations, the programmer must consider all possible flows of control through a program.

This task is hard enough if a security policy can be expressed statically, as, for instance, a list of which of a fixed set of principals is allowed to perform each of a fixed set of actions. However, the needs of real applications tend to force use of evolving security policies, and usually the most convenient place to store a policy is in the same database where the rest of application data resides. For instance, a database often encodes some kind of access control matrix, where entries reference rows of other tables. The peculiar structure of an organization may require access control based on customized schema design and checking code. An effective security validation tool must be able to "understand" these policies.

Many program analysis and instrumentation schemes have been applied to provide some automatic assurance of security properties. In this space, the traditional dichotomy is between *dynamic* and *static* tools, based on whether checking happens at run time or compile time. The two extremes have their characteristic advantages.

- Dynamic analysis can often be implemented without requiring any *program annotations* included solely to make analysis easier.
- Real developers have an easier time writing *specifications* compatible with dynamic analysis, since these specifications can often be arbitrary code for inspecting program states.
- Static analysis can provide strong guarantees that hold for all possible program executions, even those exercising weird corner cases that may not have been considered.
- Static analysis adds no run-time overhead.

In this paper, we present a tool *UrFlow* for static analysis of database-backed Web applications. We have tried to reap some of all of the advantages just described. Our tool requires no program annotations and provides fully sound static assurance about all possible executions of a program, and it requires no changes to the run-time behavior of programs. We take advantage of the fact that it is already common for Web applications to be implemented at quite a high level, relying on an SQL engine to implement the key data structures. Our tool models

the semantics of SQL faithfully, at a level that makes formal, automated analysis quite practical. We use popular ideas from symbolic execution and automated theorem-proving to build detailed models of program behavior automatically, which saves developers the trouble of explaining these models with code annotations.

It is natural for developers to write specifications that look much like the program code they are already writing. Traditional assertions (e.g., with the C `assert` macro) fall under this heading. In an application that depends on an SQL engine to manage its main data structures, it seems similarly natural to express security policies using SQL. Our tool is based on that model, allowing developers to write detailed statically-checkable specifications without learning a new language. Queries can express confidentiality properties by *selecting which information the user may learn*, and queries can express database update properties by *selecting allowable state transitions*. We need only one extension to the standard SQL syntax and semantics: to allow policies to vary by user, we introduce explicit consideration of *which secrets (e.g., passwords) the user knows*.

UrFlow is integrated with the compiler for Ur/Web [3], a domain-specific language for Web application development. Ur/Web presents a very high-level view of the domain, with explicit language support for the key elements of Web applications. For instance, the SQL interface uses an expressive type system to ensure that any code that type-checks accesses the SQL database correctly. In the present project, we have used the first-class SQL support to avoid the need for program analysis to recover a high-level view of how an application uses the database.

We begin by introducing our policy model and demonstrating its versatility. After that, we present our program analysis, including its symbolic evaluation and automated theorem-proving aspects. Next, we discuss the scope and limitations of our analysis, describe some case-study applications that we have checked with UrFlow, and compare with related work.

2 SQL Queries as Policies

Consider a simple application that maintains a database of users and per-user secret strings. We can declare our schema to Ur/Web with `table` declarations. Following standard practice in relational databases, each table includes a unique integer ID, which provides a convenient handle to pass to row-specific operations. Besides an ID, a `user` record contains a username and password, and a `secret` record contains the owning user ID and the data value.

```
table user : { Id : int, Nam : string,
               Pass : string }

table secret : { Id : int, User : int,
                 Data : string }
```

We also declare an HTTP cookie, which acts like a typed global variable which exists separately on each Web browser. This cookie tracks the authentication information for the currently logged-in user. While a more realistic program would probably rely on unique session IDs, here we adopt the less secure strategy of storing a user ID and password pair in each cookie, to simplify the example.

```
cookie login : { Id : int, Pass : string }
```

We can write a function that checks this cookie and returns its user ID if the password is correct. The code is written in a functional style, where we collapse "expressions" and "statements" into a single syntactic class. Thus, instead of determining the function return value with explicit `return` statements, we just say that the function result is the value of the single expression that is the function body.

Ur/Web code makes a lot of use of *tagged unions*, a safe analogue to C unions that is popular in functional programming languages. A tagged union value is either a simple tag, which is like an `enum` value in C; or a pairing of a tag and another value, which is like a C union, but with a convention to ensure that it is always possible to inspect a value and determine which union alternative is being used. For tag `T`, a simple tag expression is written like `T`, while the pairing of that tag with expression `e` is written `T(e)`. For instance, instead of allowing every object type to be inhabited by a special value `null`, we instead represent `null` with an explicit tag `None`, and we represent non-null object `o` as `Some(o)`. A *pattern-matching* construct `case` is used to deconstruct tagged union values.

Here is the code for a function to check the correctness of the information in the `login` cookie. It is written in a compiler intermediate language in which some higher-order functional programming idioms have been replaced with more standard imperative code.

```
fun userId() =
  case getCookie(login) of
    None => None
  | Some(li) =>
    let b = query
      (SELECT COUNT(*) > 0 AS B
       FROM user
       WHERE user.Id = {li.Id}
         AND user.Pass = {li.Pass})
      (r acc => r.B) False in
    if b then
      Some(li.Id)
    else
      error("Wrong user ID or password!")
```

Our `userId` function begins by retrieving the current value of the `login` cookie. This will either be `None`, if no value of the cookie is set; or `Some(li)`, if the ID/password record `li` has been set as the cookie value. If the cookie is not set, there is no user ID to return. Otherwise, we must consult the database to see if the password is correct.

We have literal SQL syntax embedded in the code, with splicing of variable values using curly braces. The query checks if there are any rows in the `user` table matching the cookie contents. In this intermediate language, every database read is expressed as a loop over the results of a query. The body of the loop is written as an expression with two explicitly-named new local variables: `r`, the latest row to process; and `acc`, an accumulator that is modified as we process rows. The body expression after the `=>` determines the new accumulator value after every iteration. We give `False` as the initial accumulator value. In our example here, the loop body ignores the accumulator, and we simply project the one field of any result row to save as the accumulator. The `error` function aborts program execution with an error message, which we do here when the user provides invalid credentials.

We can write the main entry point of our application to display all of the logged-in user's secrets.

```
fun main() =
  case userId() of
    None => write("You're not logged in.")
  | Some(u) =>
    query (SELECT secret.Id, secret.Data
           FROM secret
           WHERE secret.User = {u})
      (r acc =>
        write("<li> <i>");
        write(toString(r.Secret.Id));
        write("</i>: ");
        write(escape(r.Secret.Data));
        write("</li>")) ()
```

In this query loop, the accumulator is still ignored, and in fact we execute the function body solely for its side effects, which involve writing HTML to be sent to the client.

We would like to verify that this application satisfies a reasonable confidentiality policy. Intuitively, every cell of the database belongs to a particular user. We want to ensure that no user is able to read cells belonging to a different user. This simple policy expresses our intent for the cells of the `user` table.

```
policy sendClient (SELECT *
  FROM user
  WHERE known(user.Pass))
```

The informal meaning of this policy is that the user may learn any value that could be returned from this query. Every `policy` statement is followed by a keyword naming a kind of policy. In this case, that keyword is `sendClient`, which is used for confidentiality policies. Specifically, the user may learn anything about any row of `user` *whose password he knows*. The new predicate `known` models which information the client is already aware of. We assume the client knows the text of the program and the text of the HTTP request it sent. In our example, when we disclose any secret information, we know that the user's own password is known because it came from the `login` cookie, which was part of the incoming HTTP request.

A more complicated policy allows the release of information about secrets.

```
policy sendClient (SELECT *
  FROM secret, user
  WHERE secret.User = user.Id
    AND known(user.Pass))
```

We use a join between the `secret` and `user` tables, requiring that the client demonstrate knowledge of the password for the user who owns the secret.

Our tool verifies that the application satisfies these security policies. That is, every cell of the database whose value might be disclosed could have been selected by one of these queries, based on an interpretation of `known` drawn from the HTTP request that prompted an execution.

There are several opportunities for mistakes in implementing the policy. Consider what would happen if we had implemented `userId` to always return 17. When we run the compiler, we get an error message. The compiler tells us which secret may be leaked, and (in addition to the location of the offending write) we are given a first-order logic characterization of the state of the program at the time when the leak might occur.

```
User learns: r.Secret.Data
Hypotheses: secret(x1),
  r = {Secret =
    {Id = x1.Id, Data = x1.Data}},
  x1.User = 17
```

The hypotheses are generated directly from the SQL query in `main`. The first hypothesis tells us that row `x1` is in the `secret` table. Our row variable `r` is equated with a record built by projecting the requested fields from `x1`, and the last hypothesis represents the `WHERE` clause.

In the correct implementation, UrFlow explores every static path through the program, maintaining a logical state at each point. When the analysis reaches the point that triggered the error above, we have this more informative state.

```
c = cookie/login, known(c),
c = Some(c2), user(x1),
x1.Id = c2.Id, x1.Pass = c2.Pass,
secret(x2), x2.User = c2.Id,
r = {Secret = {Id = x2.Id, Data = x2.Data}}
```

The variable `c` stands for the cookie value, which is asserted to be known to the user. The SQL query from `userId` is reflected with assertions about a variable `x1`, which is the row of `user` that must have matched the query for execution to reach this point. The confidentiality policy used a join between `secret` and `user` to describe when information on secrets may be released. The program code, on the other hand, contains no joins. UrFlow understands join semantics to the point where it is able to deduce that the above logical state implies that a join, performed as in the policy, would authorize the release of everything included in the record `r`.

2.1 What is Being Checked?

We can give a simple characterization of exactly what confidentiality property the analyzer enforces, as a function of the policy the user specifies. First, we need to define exactly what we mean by the *known* predicate. Informally, a known piece of data is something that the user is already aware of, so that no confidentiality requirement is violated by echoing back that value or another value derived from it in a predictable way. More formally, *known* is the most restrictive predicate satisfying the following rules:

1. Any constant appearing in the program text is known.
2. The initial value of every cookie is known. These cookies may have arbitrary structured types, as in the record type given to the `login` cookie in the last example.
3. The value of every explicit parameter to the application is known. For page requests generated by submission of HTML forms, this includes all form field values.
4. A record is known iff all of its fields are known.
5. For any union tag `T` (e.g., `Some` in our example), a value `v` is known iff `T(v)` is known.

We say that a value v is *allowed* in a specific database state D if there exists a `sendClient` policy that, when executed in state D, would return v as one of its outputs. We say that a value v is *built from* a set S if v is in S or can be constructed out of the elements of S by combining a subset of them with record and tagged union operations.

Now we can give a concise description of exactly what UrFlow checks. For any execution of a program that the analysis approved:

1. Whenever a `write` command sends some value v to the client, v is built from the set of values that are known or allowed.
2. Whenever the program branches based on the value v of some test expression, such that the branch chosen influences what might be sent to the client later, v is built from the set of values that are known or allowed. This prevents some *implicit flows*, where the very fact that a program reaches a particular line of code may reveal secret information. Since implicit flows are a notorious source of false alarms in information flow analysis, programmers might want to turn off this piece of checking, which would be easy to do via a compiler flag.

The same kind of characterization does not work well for ruling out implicit flows induced by SQL `WHERE` clauses, so we leave additional checking of that kind for future work. This means that a checked program may leak information about the *existence* of rows, based on tests against arbitrary SQL expressions, but the *contents* of those rows will not be leaked directly.

2.2 Authorizing Database Writes

UrFlow also checks every database modification. For example, consider this page generation function, which would be given as the action to run upon submission of an HTML form for adding a new secret.

```
fun addSecret(fields) =
  case userId() of
    None => write("You're not logged in.")
  | Some u =>
    let id = nextId() in
    dml (INSERT INTO secret (Id, User, Data)
         VALUES ({id}, {u}, {fields.Data}));
    main()
```

If we do not assert an explicit database update policy, then UrFlow rejects this program. Here is one policy that would allow the insertion:

```
policy mayInsert (SELECT *
  FROM secret AS New, user
  WHERE New.User = user.Id
    AND known(user.Pass)
    AND known(New.Data))
```

We reuse the same SQL query notation for modification policies, though the choice of `SELECT` clause is ignored, so we will always write `SELECT *`. One of the

tables in the FROM clause must be given the name New; this is the table for which we are authorizing insertion.

UrFlow only allows a row insertion if the new row could be returned by one of the mayInsert queries, in a certain sense. In checking against a particular policy query, we interpret the New relation as the universal relation, containing all possible tuples. The policy may join it with other, real database tables and perform filtering with WHERE, leading to a result set of rows that may be infinite. The insertion is permitted if the New part of one of these rows matches the values being inserted.

Our insertion policy lets any user add secrets if he associates them with his own user. We can also authorize deletions and updates, based on similar criteria.

```
policy mayDelete (SELECT *
  FROM secret AS Old, user
  WHERE Old.User = user.Id
    AND known(user.Pass))

policy mayUpdate (SELECT *
  FROM secret AS Old, secret AS New, user
  WHERE Old.User = user.Id
    AND New.User = Old.User
    AND New.Id = Old.Id
    AND known(user.Pass)
    AND known(New.Data))
```

A mayDelete policy must tag a FROM table as Old, to stand for the table being deleted from. A mayUpdate policy needs both Old and New tables, standing for the part of a table being updated and the new data being written into it. Both new policies retain the logic for checking that the client knows the password for the user whose secret is affected, and the update policy also requires that the secret ID is not changed. The insertion and update policies require that the new data value is known, which provides a simple guard against inadvertent leaking of privileged information into a part of the database that is considered to be less privileged.

3 Flexibility of Query-Based Policies

We have found that this approach to writing specifications leads to natural descriptions of many natural policies. For instance, we have implemented a simple Web message forum system. Our implementation contains a table representing an access-control list. Each entry gives a user permissions in a specific forum, at a particular numeric level of access.

```
table acl : { Forum : forumId,
              User : userId, Level : int }
```

One policy allows release of information about any message in a forum that the current user has been granted any kind of access to.

```
policy sendClient (SELECT *
  FROM message, acl, user
  WHERE acl.Forum = message.Forum
    AND acl.User = user.Id
    AND known(user.Pass))
```

Posting a new message requires access at level 2 or higher.

```
policy mayInsert (SELECT *
  FROM message AS New, user, acl
  WHERE New.User = user.Id
    AND New.Forum = acl.Forum
    AND user.Id = acl.User
    AND known(user.Pass)
    AND acl.Level >= 2
    AND known(New.Subject)
    AND known(New.Body))
```

Regular users may not delete messages from forums. This right is only granted to admins, who have access level 3 or higher. The following policy formalizes the deletion rule.

```
policy mayDelete (SELECT *
  FROM message AS Old, user, acl
  WHERE Old.Forum = acl.Forum
    AND user.Id = acl.User
    AND known(user.Pass)
    AND acl.Level >= 3)
```

Our implementation allows forums to be marked as public, in which case any visitor may read their contents. There is also another ACL table which grants users admin access to all forums. Additional policies allow information flows and updates based on these rules.

The UrFlow policy language supports access control techniques besides user accounts with passwords. For example, we have implemented a simple Web-based poll system without user accounts. Anyone may create a new poll; at that time, the creator learns a secret code that grants admin rights to the poll. That code allows him to add poll questions. After adding all of the questions, the poll creator may mark the poll as live. After that time, no further changes to the poll are allowed, and the poll is added to a list on the application's front page. Anyone may vote in a live poll, but no one may vote on a poll that is not yet live. After submitting his votes, a user receives a code that allows him to view the results of the poll. Results should never be released without first checking that the user has provided a code that matches the poll admin code or a code associated with a vote that has been cast.

The policy below controls the conditions under which a new question may be added to a poll. In particular, the question must be linked to a valid poll, the user must know the admin code for the poll, and the poll must not be live yet.

```
policy mayInsert (SELECT *
  FROM question AS New, poll
  WHERE New.Poll = poll.Id
    AND known(poll.Code)
    AND NOT poll.Live
    AND known(New.Text))
```

Anyone with a poll's admin code may update the poll only to mark it as live. This policy expresses that requirement with equality assertions between old and new values of every column besides `Live`.

```
policy mayUpdate (SELECT *
  FROM poll AS New, poll AS Old
  WHERE New.Id = Old.Id
    AND New.Nam = Old.Nam
    AND New.Code = Old.Code
    AND New.Live
    AND known(Old.Code))
```

We allow release of information about answers to a poll, whenever the user proves he already voted in that poll by providing a code associated with an appropriate answer set.

```
policy sendClient (SELECT *
  FROM answer, answers AS Other,
    answers AS Self
  WHERE answer.Answers = Other.Id
    AND Other.Poll = Self.Poll
    AND known(Self.Code))
```

We believe that this specification approach is very general, while being much more accessible to the average developer than most specification languages are. To investigate the potential for static analysis based on these specifications, we implemented the UrFlow prototype, which handles a restricted subset of all SQL queries. In particular, in both policies and programs, we only process queries containing just `SELECT`, `FROM`, and `WHERE` clauses, where the `FROM` clauses must be simple comma-separated lists of tables. We also have not implemented any analysis optimizations like procedure summaries [19], and the analysis only succeeds at understanding loops and recursion following a few simple patterns.

Perhaps surprisingly, this is enough to enable sound checking of a variety of paradigmatic Web applications. We will now describe the analysis and then argue for its effectiveness with statistics about a set of representative applications that it has validated.

4 An Outline of the Analysis

Sound program checking requires considering all possible paths of execution. Since most any non-trivial Web application can effectively follow infinitely many paths, we must apply some abstraction. In implementing UrFlow, we adopted the strategy associated with tools like ESC [10], the Extended Static Checker family.

While concrete program evaluation involves program states consisting of variable values, memory states, and so on, the kind of *symbolic evaluation* that we apply involves program states consisting of *formulas of first-order logic*. Such a formula can be thought of as *describing* concrete states, so that each abstract state may stand for infinitely many concrete states. Every basic program operation can be modeled as a *predicate transformer*. Some operations may not always be safe. In the classical setting, this may be an array dereference, where the index might be out of bounds. In our case, possibly-unsafe operations include `write` commands and database updates. No matter which setting we are in, the safety of operations is checked by associating each operation with a logical condition that implies its safety.

This gives us the outline of a sound checking procedure: Start with the abstract state "true." Explore all program paths, extending the abstract state as we go. Each time we reach an operation with safety condition C while in state S, ask an *automated theorem prover* whether $S \Rightarrow C$. The ESC projects used the Simplify prover [8] for this purpose. Today, the functionality provided by Simplify is most commonly known by the name SMT, for *satisfiability modulo theories*, and there is a rich base of tools and users in the domain of static program checking.

Our outline omits a critical element of the problem: Even after abstracting program states with formulas, there are probably still infinitely many feasible program paths. The ESC approach requires additional program annotations that can be used to finitize the path space. In the design of UrFlow, we have instead taken advantage of the control-flow simplicity of the average Web application. Many interesting applications can be implemented with just one kind of loop: iteration over writing some output for every row returned by an SQL query. Such loops effect no state changes that must be taken into account in the remainder of the program, so in a sense they have trivially inferable "loop invariants." Since loop iteration does not accumulate side effects, it is sound to *traverse each loop body just once*, which ensures that each program can be broken into a finite set of finite analysis paths.

UrFlow thus works by literal exploration of all control flow paths through a program. The next section goes into more detail on the exploration strategy, pointing out the theorem prover operations that will be required. The following section presents our implementation of those prover primitives, in an engine that extends the standard SMT approach with a few new features.

5 Symbolic Evaluation

The abstract states of UrFlow are defined in terms of a simple language of logical expressions and predicates. We write c for constants (drawn from integer, floating point, and string literals), T for union tags, x for logical variables, X for program variables, F for record field names, and R for SQL table names. The following grammar describes the syntax of program states. For a token sequence t, we write $\overline{t}$ for a comma-separated list of zero or more ts.

$$\begin{array}{rrcl} \text{Expression} & e & ::= & c \mid x \mid T(e) \mid \{\overline{F = e}\} \mid e.F \\ \text{Predicate} & p & ::= & \mathsf{known}(e) \mid R(e) \mid e = e \mid \ldots \\ \text{State} & S & ::= & (\overline{p}, \overline{X \mapsto e}) \end{array}$$

A state is a pair of a variable assignment and a set of predicates. For a particular program point, a variable assignment maps every in-scope program variable into a logical expression. The predicates are expressed only in terms of logical variables, not the program variables.

Since we inline all function calls, every execution path to analyze begins at the entry point of some function that has been registered to be called in response to a particular URL pattern. The arguments to this function stand for explicit parameters and form field values, extracted from an HTTP request. Where the function arguments are named X_i, we create an initial state $(\overline{\mathsf{known}(x_i)}, \overline{X_i \mapsto x_i})$, for fresh, distinct variables x_i. At many other points in path exploration, we will generate fresh logical variables, which we always assume to be distinct from any previously-chosen variables.

For each function, we explore all paths through it. Most program expression forms are easy to process, as they admit direct translation into logical expressions. The more interesting cases come from branching and database interaction.

Our single branching construct is `case` expressions, which test a value against a number of patterns, which may bind new variables if they match. We model `if` expressions as a special case of `case` expressions, where the patterns to match against are `true` and `false`.

As an example, consider an expression like the following:

```
case e of None => e1 | Some(X) => e2
```

If `e` is just the tag `None`, then we continue with evaluating `e1`. Otherwise, `e` is `Some v` for some `v`, and we evaluate `e2` with `X` set to `v`. To capture this with symbolic evaluation, we consider both `e1` and `e2` as starts of separate execution paths. For the `e1` case, we extend the state with the predicate $v = \mathsf{None}$, where v is the result of evaluating `e`. For the `e2` case, we choose a fresh variable x, add the variable mapping $X \mapsto x$, and add the predicate $v = \mathsf{Some}(x)$.

With `case`, it is easy to write code with exponentially many control-flow paths, but where all but a few are logically impossible. For instance, we can sequence several `case` expressions that analyze the same program variable with the same patterns. Variables are immutable, so each `case` must choose the same pattern, reducing the number of feasible paths to the number of patterns. We want our automated theorem prover to detect the infeasibility of the other paths as early as possible. Concretely, this will happen on a path where two `case`s lead to assertions like $v = \mathsf{None}$ and $v = \mathsf{Some}(x)$, on a path that assumes matching of a `None` pattern the first time and a `Some` pattern the second time. The prover knows that values built with different union tags are disjoint, so it can signal a contradiction here. Whenever a contradiction is detected at some point on a path, we can skip exploring the rest of that path.

A number of primitive operations send output to the client. The simplest of these is `write`, which appends a piece of HTML to the page being generated. UrFlow enforces that the value being sent can be constructed from known and allowable pieces of data. Recall that allowable values are those that could be produced by executing `sendClient` policies in the current database state. Consider this line of our earlier example program:

```
write(escape(r.Secret.Data));
```

The record `r` has come out of a database query. To verify that this `write` conforms to the policy, we must check that `r.Secret.Data` is known, allowable, or built from such values out of record and union operations. At this point in symbolic execution, the variable mapping will map the program variable `r` to some logical variable r, and our predicate set will be:

$$\begin{array}{l} c = \mathsf{cookie/login}, \mathsf{known}(c), c = \mathsf{Some}(c'), \mathsf{user}(x_1), \\ x_1.\mathsf{Id} = c'.\mathsf{Id}, x_1.\mathsf{Pass} = c'.\mathsf{Pass}, \\ \mathsf{secret}(x_2), x_2.\mathsf{User} = c'.\mathsf{Id}, \\ r = \{\mathsf{Secret} = \{\mathsf{Id} = x_2.\mathsf{Id}, \mathsf{Data} = x_2.\mathsf{Data}\}\} \end{array}$$

The state tells us that we know of two rows that must exist in the database: x_1 from table `user` and x_2 from table `secret`. Each of our declared confidentiality policies is phrased as a `SELECT` query whose `FROM` clause mentions one or more tables. To check if a value may be written, we need to consider ways of matching the policy queries with the logical state. The same table may be mentioned multiple times in one policy or one state, so, in general, there may be many ways to match a policy's `FROM` clause with the table predicates of a state. In UrFlow, we apply the heuristic of considering at most one matching per policy. The analysis enumerates every matching of policies with row variables, subject to that constraint.

Our running example included these two policies:

```
policy sendClient (SELECT *
  FROM user
  WHERE known(user.Pass))

policy sendClient (SELECT *
  FROM secret, user
  WHERE secret.User = user.Id
    AND known(user.Pass))
```

They can be expressed in logical form, where each is a set of predicates that, if all are true, implies the allowability of a set of values.

Predicates:	$\mathsf{user}(r_1), \mathsf{known}(r_1.\mathsf{Pass})$
Values:	$r_1.\mathsf{Id}, r_1.\mathsf{Nam}, r_1.\mathsf{Pass}$
Predicates:	$\mathsf{user}(r_1), \mathsf{secret}(r_2), \mathsf{known}(r_1.\mathsf{Pass}),$ $r_2.\mathsf{User} = r_1.\mathsf{Id}$
Values:	$r_1.\mathsf{Id}, r_1.\mathsf{Nam}, r_1.\mathsf{Pass}, r_2.\mathsf{Id},$ $r_2.\mathsf{User}, r_2.\mathsf{Data}$

Matching a policy against a state is a two-step process. First, we consider a mapping of the policy's r_i row variables to variables appearing in the state. For any table predicate $R(r_i)$ appearing in the policy, we try setting r_i to x, for any $R(x)$ appearing in the state. Once we have found a plausible mapping for every policy row variable, we apply that mapping to the remaining predicates in the policy. If the theorem prover verifies that the state implies every one of these predicates, then we have found a viable policy instantiation, and we can continue matching the remaining policies. We repeat the process to try every combination of instantiating every policy at most once.

For every set of policy instantiations, we compute the set of expressions that those policies say are fair game to write. Our running example has exactly one feasible instantiation per policy: every policy variable in `user` unifies with x_1, and every policy variable in `secret` unifies with x_2. The remaining predicates are all implied by the state. Most interestingly, we must verify that the state implies $\mathsf{known}(x_1.\mathsf{Pass})$, which follows by reasoning from this subset of the state predicates:

$$\mathsf{known}(c), c = \mathsf{Some}(c'), x_1.\mathsf{Pass} = c'.\mathsf{Pass}$$

The reasoning goes like this: Because the union value c is known, its contents c' are known, too. Because the record c' is known, its field Pass is known. That field is asserted equal to the value x_1.Pass that we want to prove known, so we are done. The theorem prover provides a complete decision procedure for reasoning chains of this kind.

Having verified correct instantiation of each policy, we arrive at this set of allowable expressions:

$$x_1.\mathsf{Id}, x_1.\mathsf{Nam}, x_1.\mathsf{Pass}, x_2.\mathsf{Id}, x_2.\mathsf{User}, x_2.\mathsf{Data}$$

We are trying to prove that the expression r.Secret.Data is allowable, which requires proving that it is equal to one of the above expressions. It turns out that our state implies that the written value equals x_2.Data, because the state contains this predicate:

$$r = \{\mathsf{Secret} = \{\mathsf{Id} = x_2.\mathsf{Id}, \mathsf{Data} = x_2.\mathsf{Data}\}\}$$

That completes the check for this `write` operation. The procedure scales to handling much more complicated cases, and we also apply the same procedure to any expression used in a branching construct, such that the result of the test influences what is written to the client. Especially in this latter case, we need to be able to reason about values that are neither known nor allowable, but that are built from such values via record and union operations. Our theorem prover handles the automation of that kind of reasoning, too.

The heart of symbolic evaluation is the treatment of database queries. Recall the form of queries, as illustrated by the main output loop of our example application.

```
query (SELECT secret.Id, secret.Data
       FROM secret
       WHERE secret.User = {u})
      (r acc => ...) ()
```

We execute an SQL query, which may contain injected program values, and loop over the result rows. An accumulator is initialized to some specified value, which here is the dummy value `()`, since we execute this loop body only for side effects. Every iteration runs the loop body with `r` bound to the latest result row and `acc` bound to the current accumulator. After an iteration, the accumulator is replaced with the value of the `...` body expression.

Traditional verification tools require manual annotation of loops with invariants, to help tame the undecidability of the program analysis problem. To avoid that cost, we designed UrFlow around some observations about the loops that appear in practice in Web applications. Most are run solely for their side effects of writing content to the client, so that there is no need to track state changes from iteration to iteration. Ur/Web variables are all immutable, so it is not even possible for them to change across iterations. Side effects are restricted to database tables and cookies, which tend not to be used in the same way that variables are used in traditional imperative languages. All this implies that a simple loop traversal strategy can be very effective: *traverse each loop body only once*.

Concretely, when we reach a query in a symbolic execution path, we consider two possible sub-paths. First,

the query may return no results, in which case we proceed taking the initial accumulator as the final value.

More interestingly, the loop may execute one or more times. We perform a quick linear pass over the body `...` to see which cookies it might set and which tables it might modify with SQL `UPDATE` or `DELETE` commands. All references to those cookies and tables are deleted from the symbolic state. Since all other aspects of concrete state are immutable, this new logical state is guaranteed to be an accurate description of the concrete state *at the beginning of any iteration of the loop*. Thus, by running the loop body with its local variables set to fresh logical variables, we consider all possible behaviors of the loop. We can continue execution afterward as if we had just executed the loop body once as normal, non-loop code. The symbolic state at loop exit can just as well stand for the last iteration of the loop as for any other iteration.

At the beginning of a loop iteration, we must enrich the logical state with predicates capturing the behavior of the query. This is best illustrated by example. Consider again the main loop of our example application. We execute its loop body with variable `r` set to r and `acc` set to some arbitrary value (since the accumulator is not referenced in the body). Assume that program variable `u` is mapped to logical variable u. We add these predicates to the logical state:

$$\mathsf{secret}(x_2), x_2.\mathsf{User} = u,$$
$$r = \{\mathsf{Secret} = \{\mathsf{Id} = x_2.\mathsf{Id}, \mathsf{Data} = x_2.\mathsf{Data}\}\}$$

Queries with joins just add more table predicates, as we have seen in the modeling of queries as policies. Larger `WHERE` conditions add additional non-table predicates. A `SELECT` clause determines which fields to project from the tables, in building the record expression to equate with r.

This basic algorithm works for most of the queries that we support. In general, UrFlow does not yet support SQL grouping or aggregation. We include one special case for queries selecting just the aggregate function `COUNT(*)`. Here, we consider that the loop body always iterates exactly once. Either the query result is 0, and we do not enrich the state with any new table information; or the result is greater than 0, and we assert that there exists some set of rows matching the conditions of the query.

To check database updates, we use a hybrid of the query and write checking. Any modification must match with an update policy, using the same matching procedure as for writes, but without the need to check allowability of a value. After an `UPDATE` or `DELETE`, we delete any state predicates mentioning the affected tables.

UrFlow also has basic support for simple recursive functions. Calls to recursive functions are effectively inlined like regular function calls, with further self-calls skipped. To make this omission sound, we analyze each recursive function to find all effects it might have on the database and cookies, and every self-call is treated as a nondeterministic modification of those parts of the state, followed by generation of an unknown return value. Further analysis allows us to abstract the initial state so that it can stand for any set of arguments that might be used at any recursion depth, such that we only preserve state information that can be shown not to vary across calls. As a result, just like for query loops, a single pass over the function body suffices to consider all possible behaviors.

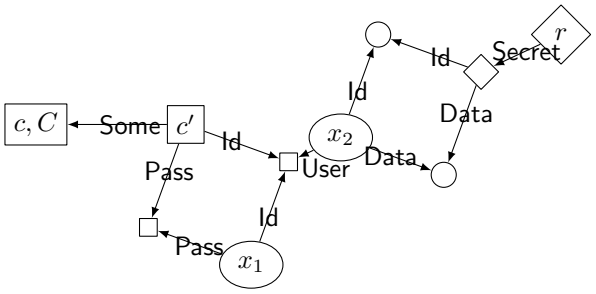


Figure 1: E-graph for the state from the `write` example

We want to emphasize some useful consequences of the way that our analysis handles SQL. First, unlike in some related work [14], despite the fact that our policies are themselves SQL queries, the analysis *does not* require that program code use exactly those queries. Semantic modeling of queries makes it possible for one policy query to justify infinitely many possible program queries. Second, the soundness of our analysis depends on knowledge of the database *schema*, but not knowledge of database *contents*. Schema changes can invalidate analysis results by, for example, redefining data integrity constraints that the theorem-prover might have relied on. However, arbitrary changes to the database rows, by arbitrary programs with no relation to UrFlow, cannot invalidate past analysis results.

6 The Theorem Prover

The last section highlighted the key theorem-prover operations that symbolic evaluation depends on. We can summarize them like this:

- Assert a predicate p. If p contradicts the predicates already asserted, raise an exception indicating so.
- Check if a predicate is implied by those already asserted.
- Determine if a logical expression can be constructed from members of a set of allowable expressions.

The first two points are supported by the classic model of first-order logic theorem-proving that is embodied in tools like Simplify [8]. The third point is new and not directly supported by usual prover interfaces, but the usual implementation techniques can support it very directly.

Provers like Simplify are based on the Nelson-Oppen architecture. We do not use many of the elements of that architecture, since our prototype implementation omits features like reasoning about arithmetic. Instead, we just adopt the key data structure, the *E-graph*. An E-graph is a directed graph representation of the possible worlds that are consistent with a set of predicates. Nodes stand for objects, and, for function symbol f, an edge labeled with f goes from node u to node v if, in any compatible world, the object associated with v equals the result of applying f to the object associated with u. A node is labeled with logical variables and constants to indicate that any compatible world must assign this node to an object equal to those variables and constants.

In UrFlow, we only use two kinds of function symbols: union tags and record field names. For tag T, there is a T-labeled edge from u to v if v must be u tagged with T (i.e., "$v = T(u)$"). For field name F, there is an F-labeled edge from u to v if u is a record whose F component equals v. For each node that came from a literal record expression, we mark that node as *complete*, in the sense that the field edges coming out of it provide a complete description of the available fields. An example of an incomplete record node is one representing a row selected in an SQL query; the state will only mention those columns relevant to the query, and it would be unsound to treat this row as if it had no further columns.

Figure 1 shows an E-graph representing the logical state given earlier for checking the code `write(escape(r.Secret.Data))`. Nodes are boxes when the state implies that they are known; other nodes may not be known. Complete record nodes are diamonds. We abbreviate `cookie/login` as `C`.

The basic prover algorithm understands two kinds of predicates: $e_1 = e_2$ and $\mathsf{known}(e)$. When either kind is asserted, its expressions are first evaluated into nodes of the E-graph, adding new nodes as necessary. A variable or constant is evaluated to the node labeled with it. A union tag application $T(e)$ is evaluated by following the T edge from the node that e evaluates to, and a field projection $e.F$ is evaluated analogously. A record expression $\{F_1 = e_1, \ldots, F_n = e_n\}$ is evaluated by checking for existing complete nodes whose F_i edges point to the nodes to which the e_is evaluate.

When a fact $e_1 = e_2$ is asserted, the nodes u_1 and u_2 standing for e_1 and e_2 are merged, taking the unions of their sets of labels and incoming and outgoing edges. Alternatively, this fact might trigger a contradiction. That happens when u_1 and u_2 are labeled with different constants or have incoming tag edges labeled with different tags.

When a fact $\mathsf{known}(e)$ is asserted, and e evaluates to u, we "change u to a box," and we propagate this knownness information across edges. That propagation follows record field edges in the forward direction only and tag edges in either direction. The same propagation is implied when merging a known node with a not-known node for an equality assertion.

The heart of the procedure is in this handling of assertion. E-graphs have nice properties which make implication checking very efficient. To check if $e_1 = e_2$, we only check if e_1 and e_2 evaluate to the same node. To check if $\mathsf{known}(e)$, we only check if e evaluates to a boxed node.

One useful addition, implemented outside of the theorem prover core, takes advantage of *key information* for SQL tables, where, for instance, an ID column is asserted not to be duplicated across rows of a table, and the SQL engine maintains this invariant with dynamic checks. Whenever a new predicate asserts that some row r is in table R, we check, for every pre-existing predicate $R(r')$, if r and r' agree on the values of R's key columns. These checks can be implemented by querying the prover core with the appropriate equality predicates. Whenever a matching r and r' pair is found, we can skip adding the new predicate $R(r)$ to the state, instead asserting $r = r'$. This enrichment of the prover is useful in analyzing applications that, for example, query a user/password table multiple times, where correctness relies on the fact that the query always returns the same result.

The last ingredient is checking if the value of expression e can be constructed out of the values of expressions $e_1, \ldots, e_n$, using only record and union operations. To implement the check, we evaluate each e_i in turn, marking its node as allowable. Next, we evaluate e to a node u. If u is marked as allowable, we are done. Otherwise, if u has an incoming union tag edge from a node v, we repeat the procedure for v. If u is a complete record node, we repeat the procedure for each target of a field edge out of u, returning success only if the check is successful for each of these new nodes. In any other case, we return failure.

7 Discussion

We can get a sense for the breadth of UrFlow by considering how it helps with the most common Web application security flaws. The OWASP Top 10 Web Application Security Risks project[1] is a popular reference for security-conscious Web developers. Based on analysis

[1] `http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`

of databases of real vulnerabilities, the OWASP team has identified which classes of security flaw pose the greatest risks. The Ur/Web compiler rules out injection (ranked #1) and cross-site scripting (#2) vulnerabilities and partially mitigates cross-site request forgery (#5) and unvalidated redirects and forwards (#10) using techniques unrelated to UrFlow. Risk #6, security misconfiguration, is a whole-system property that cannot really be addressed by any single tool, and UrFlow's lack of integrated reasoning about cryptography prevents it from helping to avoid insecure cryptographic storage (#7). UrFlow can contribute to the mitigation of the remaining risk categories.

Risk #3, broken authentication and session management, is helped by the ability to use UrFlow policies to specify exactly which secure tokens may be sent to which clients. It is still possible to make mistakes in the policies, but these policies should be significantly easier to audit than programs, with the many possible control-flow paths of the latter. The next two risk categories, insecure direct object references (#4) and failure to restrict URL access (#8), are very similar, as both involve the omission of access control checks for particular system objects. UrFlow can enforce that appropriate checks are always performed whenever database objects are used in particular ways. Insufficient transport layer protection (#9) could be avoided by adding a variant of `sendClient` policies which specifies values that may only be sent to clients over SSL connections.

Comparing against the pros and cons of security types [16], we find some interesting trade-offs. UrFlow uses high-level knowledge of programs to provide more sound reasoning without program annotations. Security-typed languages generally rely on declassification techniques where trust is granted to particular spans of code. This creates a contrast between the security-typed approach, requiring trusted code but granting soundness with respect to implicit flows; and the UrFlow approach, which requires no trusted Ur/Web functions but ignores some implicit flows. Security type annotations tend to be required throughout a program, while UrFlow avoids the need to mark up program code. However, SQL queries as policies involve some gotchas that would be less applicable to security types. For instance, it is easy to forget all or part of a policy `WHERE` clause, which has the unfortunate consequence of allowing behaviors by default.

The problem of implicit flow checking is a serious one in all kinds of information flow analysis. Where UrFlow checks implicit flows, the checking is not particularly clever, and implicit flows caused by `WHERE` clauses are ignored. Future work may be able to plug part of this hole statically, and we suspect there will also be a large role for dynamic monitoring systems, for detecting brute-force password cracking attempts and other attacks that involve many HTTP requests.

Many different logical languages have been used for specification-writing in static verification tools. We found SQL to be a convenient choice, because it is expressive enough to allow direct expression of interesting policies, and declarative enough to enable effective automated reasoning. We do not mean to claim that SQL has great expressivity or succinctness advantages over more traditional specification languages. Rather, most Web programmers are accustomed to SQL, which should help in overcoming some of the social obstacles faced in the past by attempts to get programmers to write logical specifications.

Our implementation today only handles a subset of the common SQL features. We omit support for outer joins. These should be easy to model via disjunctive formulas, covering all the possible cases of whether a row matching the join condition exists in a table, though a naive realization of this idea would probably have poor performance consequences for the theorem-prover. Grouping and aggregation are harder to encode in the quantifier-free first-order logic that we are employing. We suspect that most real programs can be checked with conservative encodings of aggregation, where we model aggregate function values as unknowns. Alternatively, we can restrict reasoning about aggregate functions to simple syntactic pattern-matching against policies. That approach also seems most practical for handling of the SQL `EXCEPT` operator, which implements a kind of negative reasoning about which rows do not exist. This is needed to write down policies like (for a conference management system) "reviewer A may see the reviews for paper B only if A *does not* have a conflict with B."

More advanced policies might also need to include non-trivial program code. For instance, a custom hashing or encryption scheme might be used. Here we encounter a common situation for static verification, where it is always possible to expand the reach of your theorem-prover to handle new program features. No single implementation will ever be able to handle all realistic programs, but we suspect that very good coverage will be possible, after the incorporation of significant practical experience with the tool.

8 Evaluation

The UrFlow prototype is implemented in about 2200 lines of Standard ML code. We have used the analysis to check a number of Ur/Web applications. There is a live demo of the applications, with links to syntax-highlighted source code, at:

`http://www.impredicative.com/ur/scdv/`

Application	Program (LoC)	Policies (LoC)	Check (sec)
Secret	138	24	0.02
Poll	196	50	0.035
User DB	84	8	-
Calendar	255	46	0.28
Forum	412	134	17.68
Gradebook	342	61	1.49

Figure 2: Lines-of-code breakdown in case studies, with time required to check the code with UrFlow

Our case studies include *Secret*, a minimal application for storing secrets that may later be retrieved via password authentication, which was used as the model for this paper's first set of running examples; the *Forum* and *Poll* applications from which Section 3's examples were drawn; a *Gradebook* application, for managing a database of student grades in courses; and a reimplementation of the *Calendar* application from the paper [5] that introduced the SIF system for combined static and dynamic checking of information flow in Web applications. Calendar, Forum, and Gradebook share a common user authentication component.

The Calendar application lets users save details of their schedules on the Web, with controlled sharing of information. By default, no one may learn anything about an event. The creator of an event may learn everything about it, and the creator may add invitees who inherit the same read privileges. The creator may also authorize users to know only the time of an event, so that those users see that time slot only as "busy" on the creator's calendar. Only event creators may modify any state related to their events.

The Gradebook application is based on a database of courses and assignments of users to be instructors, teaching assistants (TAs), or students in courses. Each student membership record contains an optional grade. Only system administrators may create courses and modify instructor lists. Instructors may set grades and control TA assignments. A TA may view all of the state associated with a course, but may not modify it. A student may view his own grades, and a student in a course may only affect that course's part of the database by dropping the course.

Figure 2 gives the number of lines in code in each of these components. An application's code is separated into the program itself and the policies. The figures here make "policy overhead" appear bigger than it would probably be in production applications, since our case studies include minimal code dedicated to providing fancy user interfaces. Still, these numbers compare favorably to those for systems like SIF, where Calendar requires 1779 lines of code. While we have a similar ratio of program to annotation, our annotations are of a different kind. 443 lines of the SIF version include annotations, in the form of security types [20] and explicit downgradings. The latter involve annotations that effectively say "the owner of a piece of information trusts this span of code, so let that span release derived information that would not otherwise be allowed." The SIF Calendar case study includes 17 such downgrades.

The UrFlow approach is very different. As no annotations are required in programs, there is no need to accept any part of a program as trusted. All checking is with respect to the declarative specification provided by the policy queries.

Our analysis detects flaws similar to those that occur frequently in real deployed systems. For instance, we examined reports for July 2010 in the National Vulnerability Database[2]. Among the relevant issues, we found CVE-2009-4927, involving privilege escalation via a surprising setting of a specific cookie; and CVE-2010-2685 and CVE-2009-4929, which allow administrative actions to be taken without proper credentials, via hand-crafted HTTP requests. UrFlow makes it easy to catch these problems, since it is not necessary to enumerate all possible attack vectors, thanks to policies that talk directly about underlying resources. For instance, we introduced a bug in the Gradebook application to mimic the cookie bug above, where we allow anyone to set any student's grade if a particular cookie is set to 1. The compiler complains that the database update policy may be violated, referencing the exact span of source code where the offending `UPDATE` statement occurs. The same output appears if we simulate a forgotten access control check, in the style of the second two issues above, by commenting out an important `if` test.

UrFlow also requires no change to the runtime behavior of a program, and this baseline performance level is greater than for most popular Web languages and frameworks, thanks to the general-purpose and domain-specific optimizations performed by the Ur/Web compiler. We present the performance of the UrFlow analysis itself in Figure 2, for runs on a Linux machine with dual 1 GHz AMD64 processors with 2 GB of RAM. Of our case studies, only Forum takes much longer than a second to check. This is because Forum has a complicated main function, with many security checks. Many different actions call the main function after performing some database modification. Every such call is analyzed afresh, as if the main function had been inlined. Techniques like procedure summaries [19] should make it possible to reduce this time significantly.

[2] `http://nvd.nist.gov/`

Very precise, logic-based program analyses often exhibit bad scaling behavior. There is no theoretical reason that UrFlow would not run into the same problems. Many programs with exponentially many feasible paths will indeed trigger exponential behavior in any realization of our algorithm. Simple experiments with parameterized families of programs also show that our current implementation produces exponential running time (with small constant factors) even on some examples that can probably be reduced to linear running time with more optimization. For instance, we tested programs made up of `if`-trees that perform the same SQL query at each of the tree's exponentially-many nodes. Primary key information implies that the `if` test always goes the same way, ruling out all but two paths through the tree. Still, exponential time usage results from our heuristic of considering two execution paths starting at each query, for the cases of zero or more than zero result rows. Much future work remains in smarter detection of redundant paths.

9 Related Work

The BAN logic [2] is a formal system for reasoning about knowledge in distributed system protocols. The rules of the logic model important aspects like transitive trust and cryptography. The spi calculus [1] pursues similar goals, introducing an explicit formalization of programs, rather than just of the knowledge that principals have at points throughout a protocol. Our known predicate is modeled on notions introduced in that line of work.

Security types [20] are a technique for static checking of information flow based on explicit data labels such as "high security" and "low security." The JFlow [15] and Jif [16] systems are realistic implementations of security typing for Java. SIF [5] extends Jif for the Web application domain. This line of work enables checking of a much broader range of applications than UrFlow can handle. By focusing on a narrow domain that naturally supports declarative implementation techniques, UrFlow is able to do sound checking without requiring any program annotations. Jif-based systems require many annotations, including explicit granting of trust to particular spans of code. The Swift system [4] extends this approach to do automatic, secure partitioning of Web application code across client and server, based on information-flow constraints.

Li and Zdancewic [14] presented a system for static checking of information-flow properties for database-backed Web applications. Their design requires that the application be programmed in terms of fixed sets of query templates with holes to be filled with different values on different invocations. Every template is annotated with security typing information for each input and output. In contrast, UrFlow infers the security-relevant characteristics of queries from a declarative policy. One policy may be enough to imply the sensitivity of outputs from many different query forms. UrFlow also applies theorem-proving technology to allow sound checking of more programs, including those where policies vary dynamically based on database contents.

Asbestos [9] and HiStar [23] are operating systems with support for dynamic enforcement of the Decentralized Information Flow Control model, which specifies which run-time flows between sensitive objects to allow. The Flume system [13] implements similar functionality on top of standard UNIX abstractions. All of these systems can support complex system architectures that fall outside the specialized orientation of UrFlow. Flume has been used to build a secured version of the MoinMoin wiki application. This port to Flume required about 1000 lines of new code and 1000 lines of modifications, and a performance cost between 34% and 43% was measured, against the baseline of interpreted Python code. Our Forum case study demonstrates that UrFlow can check policies based on access control lists, which are the main property enforced in the Flume case study.

The Resin system [22] implements a much lighter-weight approach to Web application security. Instead of relying on a fixed label model, Resin allows programmers to implement their own property checks in the language in which the application is written. Policy code may tag values with policy objects, and the Resin system takes care of flowing these policies through the system and checking them at points where the application interacts with its environment. Compared to the other systems we have mentioned, including UrFlow, Resin makes it much easier to add security checking to existing applications written in popular scripting languages like PHP and Python. Resin's lightweight policy approach can also express policies that UrFlow's policy queries cannot. On the other hand, once a programmer has learned Ur/Web and used it to implement his application, UrFlow requires little annotation and brings the standard benefits of static analysis, compared to Resin and the systems mentioned in the previous paragraph: we get once-and-for-all security guarantees, without the possibility of the application being aborted because a problem is detected at run-time; and we avoid extra run-time costs, such as the 33% CPU overhead reported for a representative PHP application instrumented with Resin.

Much work on Web application security focuses on *injection attacks*, where bugs allow untrusted user input to be passed to run-time program interpreters. Solutions have employed both static [12, 21] and dynamic [11, 17] analysis. Ur/Web rules out these problems by construction, by encoding the syntax of HTML and SQL with richly-typed objects.

Rizvi et al. [18] present a technique for fine-grained

access control over SQL queries, based on the concept of *authorization views*, which are much like UrFlow's policy queries. The key difference is that authorization views are phrased in terms of variables like `$user-id` that must be filled in by some out-of-band mechanism. With UrFlow, the correctness of authentication may itself be verified, through reasoning about the `known` predicate. The technique of Rizvi et al. is applied dynamically to individual queries, where an allowability check against the current database must be run for each query. In contrast, UrFlow can prove statically that an application never uses query results inappropriately, with no modification to run-time database operation.

The SELinks system [7] extends the Links [6] Web programming language with support for static tracking of labels through trusted functions that enforce custom policies. The natural way of expressing some queries in SELinks involves mixing customized access control checks with code that should be compiled into SQL queries. The SELinks compiler handles the translation of the custom checks into stored procedures that the database engine can run during query evaluation. UrFlow follows the alternate approach of letting the programmer be explicit about the interaction of checks and queries, such that the static analysis verifies that all this has been done correctly. In general, SELinks provides a type system that makes certain types of security proofs easier, though the SELinks compiler does not carry out those proofs itself.

10 Conclusion

We have presented UrFlow, a static program analysis that verifies adherence of database-backed Web applications to security policies. These policies may vary by database state, and they are expressed as SQL queries, a convenient format for most Web programmers. UrFlow requires no program annotations and adds no run-time overhead. A key direction for future work is adaptation of UrFlow to more traditional languages, where database access is granted less of a first-class status, so that program analysis must be run to recover some information that UrFlow depends on.

Acknowledgements We would like to thank Stephen Chong, Avraham Shinnar, our shepherd Nickolai Zeldovich, and the anonymous referees for their very helpful suggestions about this project and its presentation here.

References

[1] ABADI, M., AND GORDON, A. D. A calculus for cryptographic protocols: The spi calculus. In *Proc. CCS* (1997).

[2] BURROWS, M., ABADI, M., AND NEEDHAM, R. A logic of authentication. *ACM Trans. Comput. Syst. 8*, 1 (1990), 18–36.

[3] CHLIPALA, A. Ur: Statically-typed metaprogramming with type-level record computation. In *Proc. PLDI* (2010).

[4] CHONG, S., LIU, J., MYERS, A. C., QI, X., VIKRAM, K., ZHENG, L., AND ZHENG, X. Secure web applications via automatic partitioning. In *Proc. SOSP* (2007).

[5] CHONG, S., VIKRAM, K., AND MYERS, A. C. SIF: Enforcing confidentiality and integrity in web applications. In *Proc. USENIX Security* (2007).

[6] COOPER, E., LINDLEY, S., WADLER, P., AND YALLOP, J. Links: Web programming without tiers. In *Proc. FMCO* (2006).

[7] CORCORAN, B. J., SWAMY, N., AND HICKS, M. Cross-tier, label-based security enforcement for web applications. In *Proc. SIGMOD* (2009).

[8] DETLEFS, D., NELSON, G., AND SAXE, J. B. Simplify: a theorem prover for program checking. *J. ACM 52*, 3 (2005), 365–473.

[9] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. Labels and event processes in the Asbestos operating system. In *Proc. SOSP* (2005).

[10] FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. Extended static checking for Java. In *Proc. PLDI* (2002).

[11] HALFOND, W. G. J., AND ORSO, A. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proc. ASE* (2005).

[12] HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. Securing web application code by static analysis and runtime protection. In *Proc. WWW '04* (2004).

[13] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard OS abstractions. In *Proc. SOSP* (2007).

[14] LI, P., AND ZDANCEWIC, S. Practical information-flow control in web-based information systems. In *Proc. CSFW* (2005).

[15] MYERS, A. C. JFlow: Practical mostly-static information flow control. In *Proc. POPL* (1999).

[16] MYERS, A. C., ZHENG, L., ZDANCEWIC, S., CHONG, S., AND NYSTROM, N. Jif: Java information flow, July 2001. Software release at `http://www.cs.cornell.edu/jif`.

[17] NGUYEN-TUONG, A., GUARNIERI, S., GREENE, D., SHIRLEY, J., AND EVANS, D. Automatically hardening web applications using precise tainting. In *Proc. IFIP International Information Security Conference* (2005).

[18] RIZVI, S., MENDELZON, A., SUDARSHAN, S., AND ROY, P. Extending query rewriting techniques for fine-grained access control. In *Proc. SIGMOD* (2004).

[19] SHARIR, M., AND PNUELI, A. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981, pp. 189–233.

[20] VOLPANO, D., AND SMITH, G. A type-based approach to program security. In *Proc. International Joint Conference on the Theory and Practice of Software Development* (1997).

[21] XIE, Y., AND AIKEN, A. Static detection of security vulnerabilities in scripting languages. In *Proc. USENIX Security* (2006).

[22] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving application security with data flow assertions. In *Proc. SOSP* (2009).

[23] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *Proc. OSDI* (2006).

Accountable Virtual Machines

Andreas Haeberlen, University of Pennsylvania
Paarijaat Aditya, Rodrigo Rodrigues, Peter Druschel, Max Planck Institute for Software Systems (MPI-SWS)

Abstract

In this paper, we introduce *accountable virtual machines (AVMs)*. Like ordinary virtual machines, AVMs can execute binary software images in a virtualized copy of a computer system; in addition, they can record non-repudiable information that allows auditors to subsequently check whether the software behaved as intended. AVMs provide strong accountability, which is important, for instance, in distributed systems where different hosts and organizations do not necessarily trust each other, or where software is hosted on third-party operated platforms. AVMs can provide accountability for unmodified binary images and do not require trusted hardware. To demonstrate that AVMs are practical, we have designed and implemented a prototype AVM monitor based on VMware Workstation, and used it to detect several existing cheats in Counterstrike, a popular online multi-player game.

1 Introduction

An *accountable virtual machine (AVM)* provides users with the capability to audit the execution of a software system by obtaining a log of the execution, and comparing it to a known-good execution. This capability is particularly useful when users rely on software and services running on machines owned or operated by third parties. Auditing works for any binary image that executes inside the AVM and does not require that the user trust either the hardware or the accountable virtual machine monitor on which the image executes. Several classes of systems exemplify scenarios where AVMs are useful:

- in a competitive system, such as an online game or an auction, users may wish to verify that other players do not cheat, and that the provider of the service implements the stated rules faithfully;
- nodes in peer-to-peer and federated systems may wish to verify that others follow the protocol and contribute their fair share of resources;
- cloud computing customers may wish to verify that the provider executes their code as intended.

In these scenarios, software and hardware faults, misconfigurations, break-ins, and deliberate manipulation can lead to an abnormal execution, which can be costly to users and operators, and may be difficult to detect. When such a malfunction occurs, it is difficult to establish who is responsible for the problem, and even more challenging to produce evidence that proves a party's innocence or guilt. For example, in a cloud computing environment, failures can be caused both by bugs in the customer's software and by faults or misconfiguration of the provider's platform. If the failure was the result of a bug, the provider would like to be able to prove his own innocence, and if the provider was at fault, the customer would like to obtain proof of that fact.

AVMs address these problems by providing users with the capability to detect faults, to identify the faulty node, and to produce *evidence* that connects the fault to the machine that caused it. These properties are achieved by running systems inside a virtual machine that 1) maintains a log with enough information to reproduce the entire execution of the system, and that 2) associates each outgoing message with a cryptographic record that links that action to the log of the execution that produced it. The log enables users to detect faults by replaying segments of the execution using a known-good copy of the system, and by cross-checking the externally visible behavior of that copy with the previously observed behavior. AVMs can provide this capability for any black-box binary image that can be run inside a VM.

AVMs detect integrity violations of an execution without requiring the audited machine to run hardware or software components that are trusted by the auditor. When such trusted components are available, AVMs can be extended to detect some confidentiality violations as well, such as private data leaking out of the AVM.

This paper makes three contributions: 1) it introduces the concept of AVMs, 2) it presents the design of an *accountable virtual machine monitor (AVMM)*, and 3) it demonstrates that AVMs are practical for a specific application, namely the detection of cheating in multi-player games. Cheat detection is an interesting example application because it is a serious and well-understood problem for which AVMs are effective: they can detect

a large and general class of cheats. Out of 26 existing cheats we downloaded from the Internet, AVMs can detect every single one—without prior knowledge of the cheat's nature or implementation.

We have built a prototype AVMM based on VMware Workstation, and used it to detect real cheats in Counterstrike, a popular multi-player game. Our evaluation shows that the costs of accountability in this context are moderate: the frame rate drops by 13%, from 158 fps on bare hardware to 137 fps on our prototype, the ping time increases by about 5 ms, and each player must store or transmit a log that grows by about 148 MB per hour after compression. Most of this overhead is caused by logging the execution; the additional cost for accountability is comparatively small. The log can be transferred to other players and replayed there during the game (online) or after the game has finished (offline).

While our evaluation in this paper focuses on games as an example application, AVMs are useful in other contexts, e.g., in p2p and federated systems, or to verify that a cloud platform is providing its services correctly and is allocating the promised resources [18]. Our prototype AVMM already supports techniques such as partial audits that would be useful for such applications, but a full evaluation is beyond the scope of this paper.

The rest of this paper is structured as follows. Section 2 discusses related work, Section 3 explains the AVM approach, and Section 4 presents the design of our prototype AVMM. Sections 5 and 6 describe our implementation and report evaluation results in the context of games. Section 7 describes other applications and possible extensions, and Section 8 concludes this paper.

2 Related work

Deterministic replay: Our prototype AVMM relies on the ability to replay the execution of a virtual machine. Replay techniques have been studied for more than two decades, usually in the context of debugging, and mature solutions are available [6, 15, 16, 39]. However, replay by itself is not sufficient to detect faults on a remote machine, since the machine could record incorrect information in such a way that the replay looks correct, or provide inconsistent information to different auditors.

Improving the efficiency of replay is an active research area. Remus [11] contributes a highly efficient snapshotting mechanism, and many current efforts seek to improve the efficiency of logging and replay for multi-core systems [13, 16, 28, 29]. AVMMs can directly benefit from these innovations.

Accountability: Accountability in distributed systems has been suggested as a means to achieve practical security [26], to create an incentive for cooperative behavior [14], to foster innovation and competition in the Internet [4, 27], and even as a general design goal for dependable networked systems [43]. Several prior systems provide accountability for specific applications, including network storage services [44], peer-to-peer content distribution networks [31], and interdomain routing [2, 20]. Unlike these systems, AVMs are application independent. PeerReview [21] provides accountability for general distributed systems. However, PeerReview must be closely integrated with the application, which requires source code modifications and a detailed understanding of the application logic. It would be impractical to apply PeerReview to an entire VM image with dozens of applications and without access to the source code of each. AVMs do not have these limitations; they can make software accountable 'out of the box'.

Remote fault detection: GridCop [42] is a compiler-based technique that can be used to monitor the progress and execution of a remotely executing program by inspecting periodic beacon packets. GridCop is designed for a less hostile environment than AVMs: it assumes a trusted platform and self-interested hosts. Also, GridCop does not work for unmodified binaries, and it cannot produce evidence that would convince a third party that a fault did or did not happen.

A trusted computing platform can be used to detect if a node is running modified software [17, 30]. The approach requires trusted hardware, a trusted OS kernel, and a software and hardware certification infrastructure. Pioneer [36] can detect such modifications using only software, but it relies on recognizing sub-millisecond delay variations, which restricts its use to small networks. AVMs do not require any trusted hardware and can be used in wide-area networks.

Cheat detection: Cheating in online games is an important problem that affects game players and game operators alike [24]. Several cheat detection techniques are available, such as scanning for known hacks [23, 35] or defenses against specific forms of cheating [7, 32]. In contrast to these, AVMs are generic; that is, they are effective against an entire class of cheats. Chambers et al. [9] describe another technique to detect if players lie about their game state. The system relies on a form of tamper-evident logs; however, the log must be integrated with the game, while AVMs work for unmodified games.

3 Accountable Virtual Machines

3.1 Goals

Figure 1 depicts the basic scenario we are concerned with in this paper. Alice is relying on Bob to run some software S on a machine M, which is under Bob's control. However, Alice cannot observe M directly, she can only communicate with it over the network. Our goal is to enable Alice to check whether M behaves as ex-

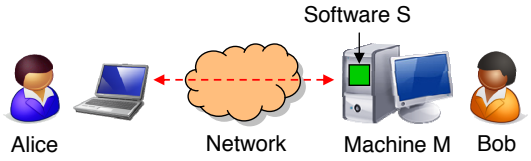


Figure 1: Basic scenario. Alice is relying on software S, which is running on a machine that is under Bob's control. Alice would like to verify that the machine is working properly, and that Bob has not modified S.

pected, without having to trust Bob, M, or any software running on M.

To define the behavior Alice expects M to have, we assume that Alice has some reference implementation of M called M_R, which runs S. We say that M is correct iff M_R can produce the same network output as M when it is started in the same initial state and given precisely the same network inputs. If M is not correct, we say that it is *faulty*. This can happen if M differs from M_R, or Bob has installed software other than S. Our goal is to provide the following properties:

- **Detection:** If M is faulty, Alice can detect this.
- **Evidence:** When Alice detects a fault on M, she can obtain evidence that would convince a third party that M is faulty, without requiring that this party trust Alice or Bob.

We are particularly interested in solutions that work for any software S that can execute on M and M_R. For example, S could be a program binary that was compiled by someone other than Alice, it could be a complex application whose details neither Alice nor Bob understand, or it could be an entire operating system image running a commodity OS like Linux or Windows.

In the rest of this paper, we will omit explicit references to S when it is clear from the context which software M is expected to run.

3.2 Approach

To detect faults on M, Alice must be able to answer two questions: 1) which exact sequence of network messages did M send and receive, and 2) is there a correct execution of M_R that is consistent with this sequence of messages? Answering the former is not trivial because a faulty M—or a malicious Bob—could try to falsify the answer. Answering the latter is difficult because the number of possible executions for any nontrivial software is large.

Alice can solve this problem by combining two seemingly unrelated technologies: tamper-evident logs and virtual machines. A *tamper-evident log* [21] requires each node to record all the messages it has sent or received. Whenever a message is transmitted, the sender and the receiver must prove to each other that they have added the message to their logs, and they must commit to the contents of their logs by exchanging an *authenticator* – essentially, a signed hash of the log. The authenticators provide nonrepudiation, and they can be used to detect when a node tampers with its log, e.g., by forging, omitting, or modifying messages, or by forking the log.

Once Alice has determined that M's message log is genuine, she must either find a correct execution of M_R that matches this log, or establish that there isn't one. To help Alice with this task, M can be required to record additional information about nondeterministic events in the execution of S. Given this information, Alice can use deterministic replay [8, 15] to find the correct execution on M_R, provided that one exists.

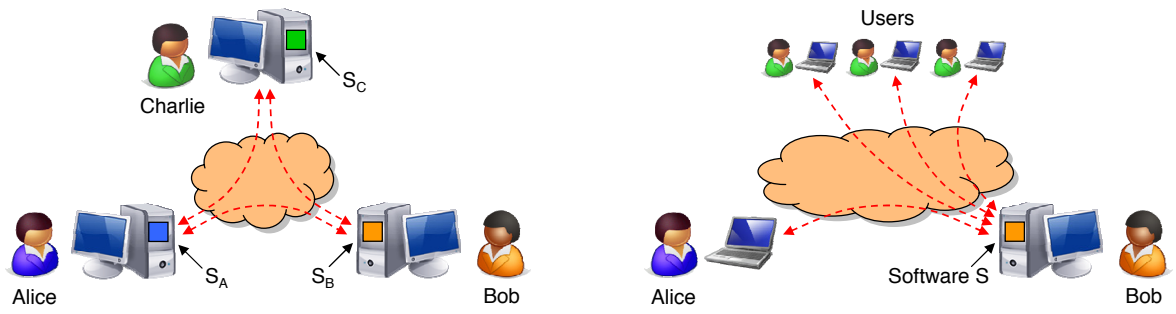
Recording the relevant nondeterministic events seems difficult at first because we have assumed that neither Alice nor Bob have the expertise to make modifications to S; however, Bob can avoid this by using a *virtual machine monitor (VMM)* to monitor the execution of S and to capture inputs and nondeterministic events in a generic, application-independent way.

3.3 AVM monitors

The above building blocks can be combined to construct an *accountable virtual machine monitor (AVMM)*, which implements AVMs. Alice and Bob can use an AVMM to achieve the goals from Section 3.1 as follows:

1. Bob installs an AVMM on his computer and runs the software S inside an AVM. (From this point forward, M refers to the entire stack consisting of Bob's computer, the AVMM running on Bob's computer, and Alice's virtual machine image S, which runs on the AVMM.)
2. The AVMM maintains a tamper-evident log of the messages M sends or receives, and it also records any nondeterministic events that affect S.
3. When Alice receives a message from M, she detaches the authenticator and saves it for later.
4. Alice periodically audits M as follows: she asks the AVMM for its log, verifies it against the authenticators she has collected, and then uses deterministic replay to check the log for faults.
5. If replay fails or the log cannot be verified against one of the authenticators, Alice can give M_R, S, the log, and the authenticators to a third party, who can repeat Alice's checks and independently verify that a fault has occurred.

This generic methodology meets our previously stated goals: Alice can detect faults on M, she can obtain evidence, and a third party can check the evidence without having to trust either Alice or Bob.



(a) Symmetric multi-party scenario (online game)

(b) Asymmetric multi-party scenario (web service)

Figure 2: Multi-party scenarios. The scenario on the left represents a multi-player game; each player is running the game client on his local machine and wants to know whether any other players are cheating. The scenario on the right represents a hosted web service: Alice's software is running on Bob's machine, but the software typically interacts with users other than Alice, such as Alice's customers.

3.4 Does the AVMM have to be trusted?

A perhaps surprising consequence of this approach is that *the AVMM does not have to be trusted by Alice*. Suppose Bob is malicious and secretly tampers with Alice's software and/or the AVMM, causing M to become faulty. Bob cannot prevent Alice from detecting this: if he tampers with M's log, Alice can tell because the log will not match the authenticators; if he does not, Alice obtains the exact sequence of observable messages M has sent and received, and since by our definition of a fault there is *no* correct execution of M_R that is consistent with this sequence, deterministic replay inevitably fails, no matter what the AVMM recorded.

3.5 Must Alice check the entire log?

For many applications, including the game we consider in this paper, it is perfectly feasible for Alice to audit M's entire log. However, for long-running, compute-intensive applications, Alice may want to save time by doing spot checks on a few log segments instead. The AVMM can enable her to do this by periodically taking a snapshot of the AVM's state. Thus, Alice can independently inspect any segment that begins and ends at a snapshot.

Spot checking sacrifices the completeness of fault detection for efficiency. If Alice chooses to do spot checks, she can only detect faults that manifest as incorrect state transitions in the segments she inspects. An incorrect state transition in an unchecked segment, on the other hand, could permanently modify M's state in a way that is not detectable by checking subsequent segments. Therefore, Alice must be careful when choosing an appropriate policy.

Alice could inspect a random sample of segments plus any segments in which a fault could most likely have a long-term effect on the AVM's state (e.g., during initialization, authentication, key generation). Or, she could inspect segments when she observes suspicious results, starting with the most recent segment and working backwards in reverse chronological order. Spot-checking is most effective in applications where the faults of interest likely occur repeatedly and a single instance causes limited harm, where the application state is frequently reinitialized (preventing long-term effects of a single undetected fault on the state), or where the threat of probabilistic detection is strong enough to deter attackers.

3.6 Do AVMs work with multiple parties?

So far, we have focused on a simple two-party scenario; however, AVMs can be used in more complex scenarios. Figure 2 shows two examples. In the scenario on the left, the players in an online multi-player game are using AVMs to detect whether someone is cheating. Unlike the basic scenario in Figure 1, this scenario is symmetric in the sense that each player is both running software *and* is interested in the correctness of the software on all the other machines. Thus, the roles of auditor and auditee can be played by different parties at different times. The scenario on the right represents a hosted web service: the software is controlled and audited by Alice, but the software typically interacts with parties other than Alice, such as Alice's customers.

For clarity, we will explain our system mostly in terms of the simple two-party scenario in Figure 1. In Section 4.6, we will describe differences for the multi-party case.

4 AVMM design

To demonstrate that AVMs are practical, we now present the design of a specific AVMM.

4.1 Assumptions

Our design relies on the following assumptions:

1. All transmitted messages are eventually received, if retransmitted sufficiently often.
2. All parties (machines and users) have access to a hash function that is pre-image resistant, second pre-image resistant, and collision resistant.
3. Each party has a certified keypair, which can be used to sign messages. Neither signatures nor certificates can be forged.
4. If a user needs to audit the log of a machine, the user has access to a reference copy of the VM image that the machine is expected to use.

The first two are common assumptions made about practical distributed systems. In particular, the first assumption is required for liveness, otherwise it could be impossible to ever complete an audit. The third assumption could be satisfied by providing each machine with a keypair that is signed by the administrator; it is needed to prevent faulty machines from creating fake identities. The fourth assumption is required so that the auditor knows which behaviors are correct.

4.2 Roadmap

Our design instantiates each of the building blocks we have described in Section 3.2: a VMM, a tamper-evident log, and an auditing mechanism. Here, we give a brief overview; the rest of this section describes each building block in more detail.

For the *tamper-evident log* (Section 4.3), we adapt a technique from PeerReview [21], which already comes with a proof of correctness [22]. We extend this log to also include the VMM's execution trace.

The *VMM* we use in this design (Section 4.4) virtualizes a standard commodity PC. This platform is attractive because of the vast amount of existing software that can run on it; however, for historical reasons, it is harder to virtualize than a more modern platform such as Java or .NET. In addition, interactions between the software and the virtual 'hardware' are much more frequent than, e.g., in Java, resulting in a potentially higher overhead.

For *auditing* (Section 4.5), we provide a tool that authenticates the log, then checks it for tampering, and finally uses deterministic replay to determine whether the contents of the log correspond to a correct execution of M_R. If the tool finds any discrepancy between the events in the log and the events that occur during replay, this indicates a fault. Note that, while events such as thread scheduling may appear nondeterministic to an application, they are in fact deterministic from the VMM's perspective. Therefore, as long as all external events (e.g. timer interrupts) are recorded in the log, even race conditions are reproduced exactly during replay and cannot result in false positives.[1]

4.3 Tamper-evident log

The tamper-evident log is structured as a hash chain; each log entry is of the form $e_i := (s_i, t_i, c_i, h_i)$, where s_i is a monotonically increasing sequence number, t_i a type, and c_i data of the specified type. h_i is a hash value that must be linked to all the previous entries in the log, and yet efficient to create. Hence, we compute it as $h_i = H(h_{i-1} \,||\, s_i \,||\, t_i \,||\, H(c_i))$ where $h_0 := 0$, H is a hash function, and $||$ stands for concatenation.

To detect when Bob's machine M forges incoming messages, Alice signs each of her messages with her own private key. The AVMM logs the signatures together with the messages, so that they can be verified during an audit, but it removes them before passing the messages on to the AVM. Thus, this process is transparent to the software running inside the AVM.

To ensure nonrepudiation, the AVMM attaches an authenticator to each outgoing message m. The authenticator for an entry e_i is $a_i := (s_i, h_i, \sigma(s_i \,||\, h_i))$, where the $\sigma(\cdot)$ operator denotes a cryptographic signature with the machine's private key. M also includes h_{i-1}, so that Alice can recalculate $h_i = H(h_{i-1} \,||\, s_i \,||\, \text{SEND} \,||\, H(m))$ and thus verify that the entry e_i is in fact $\text{SEND}(m)$.

To detect when M drops incoming or outgoing messages, both Alice and the AVMM send an *acknowledgment* for each message m they receive. Analogous to the above, M's authenticator in the acknowledgment contains enough information for the recipient to verify that the corresponding entry is $\text{RECV}(m)$. Alice's own acknowledgment contains just a signed hash of the corresponding message, which the AVMM logs for Alice. When an acknowledgment is not received, the original message is retransmitted a few times. If Alice stops receiving messages from M altogether, she can only suspect that M has failed.

When Alice wants to audit M, she retrieves a pair of authenticators (e.g., the ones with the lowest and highest sequence numbers) and challenges M to produce the log segment that connects them. She then verifies that the hash chain is intact. Because the hash function is second pre-image resistant, it is computationally infeasible to modify the log without breaking the hash chain. Thus, if M has reordered or tampered with a log entry in that segment, or if it has forked its log, M's hash chain will no longer match its previously issued authenticators, and Alice can detect this using this check.

[1] Ensuring deterministic replay on multiprocessor machines is more difficult. We will discuss this in Section 7.4.

4.4 Virtual machine monitor

In addition to recording all incoming and outgoing messages to the tamper-evident log, the AVMM logs enough information about the execution of the software to enable deterministic replay.

Recording nondeterministic inputs: The AVMM must record all of the AVM's nondeterministic inputs [8]. If an input is asynchronous, the precise timing within the execution must be recorded, so that the input can be reinjected at the exact same point during replay. Hardware interrupts, for example, fall into this category. Note that wall-clock time is not sufficiently precise to describe the timing of such inputs, since the instruction timing can vary on most modern CPUs. Instead, the AVMM uses a combination of instruction pointer, branch counter, and, where necessary, additional registers [15].

Not all inputs are nondeterministic. For example, the values returned by accesses to the AVM's virtual harddisk need not be recorded. Alice knows the system image that the machine is expected to use, and can thus reconstruct the correct inputs during replay. Also many inputs such as software interrupts are synchronous, that is, they are explicitly requested by the AVM. Here, the timing need not be recorded because the requests will be issued again during replay.

Detecting inconsistencies: The tamper-evident log now contains two parallel streams of information: Message exchanges and nondeterministic inputs. Incoming messages appear in both streams: first as messages, and then, as the AVM reads the bytes in the message, as a sequence of inputs. If Bob is malicious, he might try to exploit this by forging messages or by dropping or modifying a message that was received on M before it is injected into the AVM. To detect this, the AVMM cross-references messages and inputs in such a way that any discrepancies can easily be detected during replay.

Snapshots: To enable spot checking and incremental audits (Section 3.5), the AVMM periodically takes a snapshot of the AVM's current state. To save space, snapshots are incremental, that is, they only contain the state that has changed since the last snapshot. The AVMM also maintains a hash tree over the state; after each snapshot, it updates the tree and then records the top-level value in the log. When Alice audits a log segment, she can either download an entire snapshot or incrementally request the parts of the state that are accessed during replay. In either case, she can use the hash tree to authenticate the state she has downloaded.

Taking frequent snapshots enables Alice to perform fine-grain audits, but it also increases the overhead. However, snapshotting techniques have become very efficient; recent work on VM replication has shown that incremental snapshots can be taken up to 40 times per second [11] and with only brief interruptions of the VM, on the order of a few milliseconds. Accountability requires only infrequent snapshots (once every few minutes or hours), so the overhead should be low.

4.5 Auditing and replay

When Alice wants to audit a machine M, she performs the following three steps. First, Alice obtains a segment of M's log and the authenticators that M produced during the execution, so that the log's integrity can be verified. Second, she downloads a snapshot of the AVM at the beginning of the segment. Finally, she replays the entire segment, starting from the snapshot, to check whether the events in the log correspond to a correct execution of the reference software.

Verifying the log: When Alice wants to audit a log segment $e_i \ldots e_j$, she retrieves the authenticators she has received from M with sequence numbers in $[s_i, s_j]$. Next, Alice downloads the corresponding log segment L_{ij} from M, starting with the most recent snapshot before e_i and ending at e_j; then she verifies the segment against the authenticators to check for tampering. If this step succeeds, Alice is convinced that the log segment is genuine; thus, she is left with having to establish that the execution described by the segment is correct.

If M is faulty, Alice may not be able to download L_{ij} at all, or M could return a corrupted log segment that causes verification to fail. In either case, Alice can use the most recent authenticator a_j as evidence to convince a third party of the fault. Since the authenticator is signed, the third party can use a_j to verify that log entries with sequence numbers up to s_j must exist; then it can repeat Alice's audit. If no reply is obtained, Alice will suspect Bob.

Verifying the snapshot: Next, Alice must obtain a snapshot of the AVM's state at the beginning of the log segment L_{ij}. If Alice is auditing the entire execution, she can simply use the original software image S. Otherwise she downloads a snapshot from M and recomputes the hash tree to authenticate it against the hash value in L_{ij}.

Verifying the execution: For the final step, Alice needs three inputs: The log segment L_{ij}, the VM snapshot, and the public keys of M and any users who communicated with M. The audit tool performs two checks on L_{ij}, a syntactic check and a semantic check. The syntactic check determines whether the log itself is well-formed, whereas the semantic check determines whether the information in the log corresponds to a correct execution of M_R.

For the syntactic check, the audit tool checks whether all log entries have the proper format, verifies the cryptographic signatures in each message and acknowledgment, checks whether each message was acknowledged, and checks whether the sequence of sent and received

messages corresponds to the sequence of messages that enter and exit the AVM. If any of these tests fail, the tool reports a fault.

For the semantic check, the tool locally instantiates a virtual machine that implements M_R, and it initializes the machine with the snapshot, if any, or S. Next, it reads L_{ij} from beginning to end, replaying the inputs, checking the outputs against the outputs in L_{ij}, and verifying any snapshot hashes in L_{ij} against snapshots of the replayed execution (to be sure that the snapshot at the end of L_{ij} is also correct). If there is any discrepancy whatsoever (for example, if the virtual machine produces outputs that are not in the log, or if it requests the synchronous inputs in a different order), replay terminates and reports a fault. In this case, Alice can use L_{ij} and the authenticators as evidence to convince Bob, or any other interested party, that M is faulty.

If the log segment L_{ij} passes all of the above checks, the tool reports success and then terminates. Auditing can be performed offline (after the execution of a given log segment is finished) or online (while the execution is in progress).

4.6 Multi-party scenario

So far, we have described the AVMM in terms of the simple two-party scenario. A multi-party scenario requires three changes. First, when some user wants to audit a machine M, he needs to collect authenticators from other users that may have communicated with M. In the gaming scenario in Figure 2(a), Alice could download authenticators from Charlie before auditing Bob. In the web-service scenario in Figure 2(b), the users could forward any authenticators they receive to Alice.

Second, with more than two parties, network problems could make the same node appear unresponsive to some nodes and alive to others. Bob could exploit this, for instance, to avoid responding to Alice's request for an incriminating log segment, while continuing to work with other nodes. To prevent this type of attack, Alice forwards the message that M does not answer as a *challenge* for M to the other nodes. All nodes stop communicating with M until it responds to the challenge. If M is correct but there is a network problem between M and Alice, or M was temporarily unresponsive, it can answer the challenge and its response is forwarded to Alice.

Third, when one user obtains evidence of a fault, he may need to distribute that evidence to other interested parties. For example, in the gaming scenario, if Alice detects that Bob is cheating, she can send the evidence to Charlie, who can verify it independently; then both can decide never to play with Bob again.

4.7 Guarantees

Given our assumptions from Section 4.1 and the fault definition from Section 3.1, the AVMM offers the following two guarantees:

- **Completeness:** If the machine M is faulty, a full audit of M will report a fault and produce evidence against M that can be verified by a third party.
- **Accuracy:** If the machine M is *not* faulty, no audit of M will report a fault, and there cannot exist any valid evidence against M.

If Alice performs spot checks on a number of log segments $s_1, \ldots, s_k$ rather than a full audit, accuracy still holds. However, if M is faulty, her audit will only report the fault and produce evidence if there exists at least one log segment s_i in which the fault manifests. These guarantees are independent of the software S, and they hold for any fault that manifests as a deviation from M_R, even if Alice, Bob, and/or other users are malicious. A proof of these properties is presented in a separate technical report [19].

Since our design is based on the tamper-evident log from PeerReview [21], the resulting AVMM inherits a powerful property from PeerReview: in a distributed system with multiple nodes, it is possible to audit the execution of the entire system by auditing each node individually. For more details, please refer to [21].

4.8 Limitations

We note two limitations implied by the AVMM's guarantees. First, AVMs cannot detect bugs or vulnerabilities in the software S, because the expected behavior of M is defined by M_R and thus S. If S has a bug and the bug is exercised during an execution, an audit will succeed. For instance, if S allows unauthorized software modifications, Bob could use this feature to change or replace S. Alice must therefore make sure that S does not have vulnerabilities that Bob could exploit.

Second, any behavior that can be achieved by providing appropriate inputs to M_R is considered correct. When such inputs come from sources other than the network, they cannot be verified during an audit. In some applications, Bob may be able to exploit this fact by recording local (non-network) inputs in the log that elicit some behavior in M_R he desires.

5 Application: Cheat detection in games

AVMs and AVMMs are application-independent, but for our evaluation, we focus on one specific application, namely cheat detection. We begin by characterizing the class of cheats that AVMs can detect, and we discuss how AVMs compare to the anti-cheat systems that are in use today.

5.1 How are cheats detected today?

Today, many online games use anti-cheating systems like PunkBuster [35], the Warden [23] or Valve Anti-Cheat (VAC) [38]. These systems work by scanning the user's machine for known cheats [23, 24, 35]; some allow the game admins to request screenshots or to perform memory scans. In addition to privacy concerns, this approach has led to an arms race between cheaters and game maintainers, in which the former constantly release new cheats or variations of existing ones, and the latter must struggle to keep their databases up to date.

5.2 How can AVMs be used with games?

Recall that AVMs run entire VM images rather than individual programs. Hence, the players first need to agree on a VM image that they will use. For example, one of them could install an operating system and the game itself in a VM, create a snapshot of the VM, and then distribute the snapshot to the other players. Each player then initializes his AVM with the agreed-upon snapshot and plays while recording a log. If a player wishes to reassure himself that other players have not cheated, he can request their logs (during or after the game), check them for tampering, and replay them using his own, trusted copy of the agreed-upon VM image.

Since many cheats involve installing additional programs or modifying existing ones, it is important to disable software installation in the snapshot that is used during the game, e.g., by revoking the necessary privileges from all accounts that are accessible to the players. Otherwise, downloading and installing a cheat would simply be re-executed during replay without causing any discrepancies. However, note that this restriction is only required *during* the game; it does not prevent the maintainer of the original VM image from installing upgrades or patches.

5.3 How do players cheat in games?

Players can cheat in many different ways – a recent taxonomy [41] identified no less than fifteen different types of cheats, including collusion, denial of service, timing cheats, and social engineering. In Section 5.4, we discuss which of these cheats AVMs are effective against, and we illustrate our discussion with three concrete examples of cheats that are used in Counterstrike. Since the reader may not be familiar with these cheats, we describe them here first.

The first cheat is an *aimbot*. Its purpose to help the cheater with target acquisition. When the aimbot is active, the cheater only needs to point his weapon in the approximate direction of an opponent; the aimbot then automatically aims the weapon exactly at that opponent. An aimbot is an example of a cheat that works, at least conceptually, by feeding the game with forged inputs.

The second cheat is a *wallhack*. Its purpose is to allow the cheater to see through opaque objects, such as walls. Wallhacks work because the game usually renders a much larger part of the scenery than is actually visible on screen. Thus, if the textures on opaque objects are made transparent or removed entirely, e.g., by a special graphics driver [37], the objects behind them become visible. A wallhack is an example of a cheat that violates secrecy; it reveals information that is available to the game but is not meant to be displayed.

The third cheat is *unlimited ammunition*. The variant we used identifies the memory location in the Counterstrike process that holds the cheater's current amount of ammunition, and then periodically writes a constant value to that location. Thus, even if the cheater constantly fires his weapon, he never runs out (similar cheats exist for other resources, e.g., unlimited health). This cheat changes the network-visible behavior of the cheater's machine. It is representative of a larger class of cheats that rely on modifying local in-memory state; other examples include teleportation, which changes the variable that holds the player's current position, or unlimited health.

5.4 Which cheats can AVMs detect?

AVMs are effective against two specific, broad classes of cheats, namely

1. cheats that need to be installed along with the game in some way, e.g., as loadable modules, patches, or companion programs; and
2. cheats that make the network-visible behavior of the cheater's machine inconsistent with any correct execution.

Both types of cheats cause replay to fail when the cheater's machine is audited. In the first case, the reason is that replay can only succeed if the VM images used during recording and replay produce the same sequence of events recorded in the log. If different code is executed or different data is read at any time, replay almost certainly diverges soon afterward. In the second case, replay fails by definition because there exists *no* correct execution that is consistent with the network traffic the cheater's machine has produced.

If a cheat is in the first class but not in the second, it may be possible to re-engineer it to avoid detection. Common examples include cheats that violate secrecy, such as wallhacks, and cheats that rely on forged inputs, such as aimbots. For instance, a cheater might implement an aimbot as a separate program that runs outside

Total number of cheats examined	26
Cheats detectable with AVMs	26
... in this specific implementation of the cheat	22
... no matter how the cheat is implemented	4
Cheats not detectable with AVMs	0

Table 1: Detectability of Counterstrike cheats from popular Counterstrike discussion forums

of the AVM and aims the player's weapon by feeding fake inputs to the AVM's USB port. A particularly tech-savvy cheater might even set up a second machine that uses a camera to capture the game state from the first machine's screen and a robot arm to type commands on the first machine's keyboard. While such cheats are by no means impossible, they do require substantially more effort and expertise than a simple patch or module that manipulates the game state directly. Thus, AVMs raise the bar significantly for such cheats.

In contrast, cheats in the second class can be detected by AVMs in *any* implementation. Examples of such cheats include unlimited ammunition, unlimited health, or teleportation. For instance, if a player has k rounds of ammunition and uses a cheat of any type to fire more than k shots, replay inevitably fails because there is *no* correct execution of the game software in which a player can fire after having run out of ammunition. AVMs are effective against any current or future cheats that fall into this category.

We hypothesize that the first class includes almost all cheats that are in use today. To test this hypothesis, we downloaded and examined 26 real Counterstrike cheats from popular discussion forums on the Internet (Table 1). We found that every single one of them had to be installed in the game AVM to be effective, and would therefore be detected. We also found that at least 4 of the 26 cheats additionally belonged to the second class and could therefore be detected not only in their current form, but also in any future implementation.

5.5 Summary

Even though we did not specifically design AVMs for cheat detection, they do offer three important advantages over current anti-cheating solutions like VAC or PunkBuster. First, they protect players' privacy by separating auditable computation (the game in the AVM) from non-auditable computation (e.g., browser or banking software running outside the AVM). Second, they are effective against virtually all current cheats, including novel, rare, or unknown cheats. Third, they are guaranteed to detect all possible cheats of a certain type, no matter how they are implemented.

6 Evaluation

In this section, we describe our AVMM prototype, and we report how we used it to detect cheating in Counterstrike, a popular multi-player game. Our goal is to answer the following three questions:

1. Does the AVMM work with state-of-the-art games?
2. Are AVMs effective against real cheats?
3. Is the overhead low enough to be practical?

6.1 Prototype implementation

Our prototype AVMM implementation is based on VMware Workstation 6.5.1, a state-of-the-art virtual machine monitor whose source code we obtained through VMware's Academic Program. VMware Workstation supports a wide range of guest operating systems, including Linux and Microsoft Windows, and its VMM already supports many features that are useful for AVMs, such as deterministic replay and incremental snapshots. We extended the VMM to record extra information about incoming and outgoing network packets, and we added support for tamper-evident logging, for which we adapted code from PeerReview [21]. Since VMware Workstation only supports uniprocessor replay, our prototype is limited to AVMs with a single virtual core (see Section 7.4 for a discussion of multiprocessor replay). However, most of the logging functionality is implemented in a separate daemon process that communicates with the VMM through kernel-level pipes, so the AVMM can take advantage of multi-core CPUs by using one of the cores for logging, cryptographic operations and auditing, while running AVMs on the other cores at full speed.

Our audit tool implements a two-step process: Players first perform the syntactic check using a separate tool and then run the semantic check by replaying the log in a local AVM, using a copy of the VM image they trust. If at least one of the two stages fails, they can give the log and the authenticators as evidence to fellow players—or, indeed, any third party. All steps are deterministic, so the other party will obtain the same result.

6.2 Experimental setup

For our evaluation, we used the AVMM prototype to detect cheating in Counterstrike. There are two reasons for this choice. First, Counterstrike is played in a variety of online leagues, as well as in worldwide championships such as the World Cyber Games, which makes cheating a matter of serious concern. Second, there is a large and diverse ecosystem of readily available Counterstrike cheats, which we can use for our experiments.

Our experiments are designed to model a Counterstrike game as it would be played at a competition or

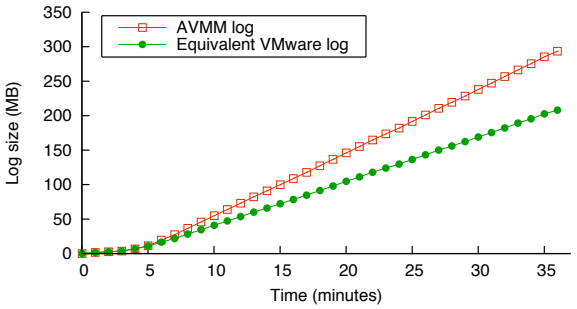


Figure 3: Growth of the AVMM log, and an equivalent VMware log, while playing Counterstrike.

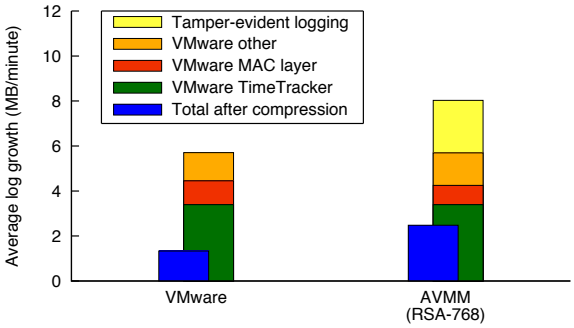


Figure 4: Average log growth for Counterstrike by content. The bars in front show the size after compression.

at a LAN party. We used three Dell Precision T1500 workstations, one for each player, with 8 GB of memory and 2.8 GHz Intel Core i7 860 CPUs. Each CPU has four cores and two hyperthreads per core. The machines were connected to the same switch via 1 Gbps Ethernet links, and they were running Linux 2.6.32 (Debian 5.0.4) as the host operating system. On each machine, we installed an AVMM binary that was based on a VMware Workstation 6.5.1 release build. Each player had access to an 'official' VM snapshot, which contained Windows XP SP3 as the guest operating system, as well as Counterstrike 1.6 at patch version 1.1.2.5. Sound and voice were disabled in the game and in VMware. As discussed in Section 5.2, we configured the snapshot to disallow software installation. In the snapshot, the OS was already booted, and the player was logged in without administrator privileges.

All players were using 768-bit RSA keys. These keys are not strong enough to provide long-term security, but in our scenario the signatures only need to last until any cheaters have been identified, i.e., at most a few days or weeks beyond the end of the game. In December 2009, factoring a 768-bit number took almost 2,000 Opteron-CPU years [3], so this key length should be safe for gaming purposes for some time to come.

To quantify the costs of various aspects of AVMs, we ran experiments in five different configurations. **bare-hw** is our baseline configuration in which the game runs directly on the hardware, without virtualization. **vmware-norec** adds the virtual machine monitor without our modifications, and **vmware-rec** adds the logging for deterministic replay. **avmm-nosig** uses our AVMM implementation without signatures, and **avmm-rsa768** is the full system as described.

We removed the default frame rate cap of 72 fps, so that Counterstrike rendered frames as quickly as the available CPU resources allow and we can use the achieved frame rate as a performance metric. In Section 6.5 we consider a configuration with the default frame rate cap. To make sure the performance of bare-hw and virtualized configurations can be compared, we configured the game to run without OpenGL, which is not supported in our version of VMware Workstation, and we ran the game in window rather than full-screen mode. We played each game for at least thirty minutes.

6.3 Functionality check

Recall from Section 5.4 that AVMs can detect by design all of the 26 cheats we examined. As a sanity check to validate our implementation, we tried four Counterstrike cheats in our collection that do not depend on OpenGL. For each cheat, we created a modified VM image that had the cheat preinstalled, and we ran an experiment in the avmm-rsa768 configuration where one of the players used the special VM image and activated the cheat. We then audited each player; as expected, the audits of the honest players all succeeded, while the audits of the cheater failed due to a divergence during replay.

6.4 Log size and contents

The AVMM records a log of the AVM's execution during game play. To determine how fast this log grows, we played the game in the avmm-rsa768 configuration, and we measured the log size over time. Figure 3 shows the results. The log grows slowly while players are joining the game (until about 3 minutes into the experiment) and then continues to grow steadily during game play, by about 8 MB per minute. For comparison, we also show the size of an equivalent VMware log; the difference is due to the extra information that is required to make the log tamper-evident.

Figure 4 shows the average log growth rate about the content. More than 70% of the AVMM log consist of information needed for replay; tamper-evident logging is responsible for the rest. The replay information consists mainly of `TimeTracker` entries (59%), which are used by the VMM to record the exact timing of events, and MAC-layer entries (14%), such as incom-

ing or outgoing network packets; other entry types account for the remaining 27%. The composition of the VMware log differs slightly because the packet payloads are stored in the MAC-layer entries rather than in the tamper-evident logging entries. We also show results after applying `bzip2` and a lossless, VMM-specific (but application-independent) compression algorithm we developed. This brings the average log growth rate to 2.47 MB per minute.

From these results, we can estimate that a one-hour game session would result in a 480 MB log, or 148 MB after compression. Thus, given that current hard disk capacities are measured in terabytes, storage should not be a problem, even for very long games. Also, when a player is audited, he must upload his log to his fellow players. If the game is played over the Internet, uploading a one-hour log would take about 21 minutes over a 1 Mbps upstream link. If the game is played over a LAN, e.g., at a competition, the upload would complete in a few seconds. To avoid detection delays, our prototype can also perform auditing concurrently with the game; we evaluate this feature in Section 6.11.

6.5 Low growth with the frame rate cap

Recall that Counterstrike was configured without a frame rate cap in our experiments, so that the measured frame rate can be used as a performance metric. We discovered that when the frame rate cap is enabled, Counterstrike appears to implement inter-frame delays by busy-waiting in a tight loop, reading the system clock. Since the AVMM has to log every clock access as a nondeterministic input, this increases the log growth considerably—by a factor of 18 when the default cap of 72 fps is used.

To reduce the log growth for applications that exhibit this behavior, we experimented with the following optimization. Whenever the AVMM observes consecutive clock reads from the same AVM within $5\,\mu$s of each other, it delays the n.th consecutive read by $2^{n-2}*50\,\mu$s, starting with the second read and up to a limit of 5 ms. The exponential progression of delays limits the number of clock reads during long waits, but does not unduly affect timing accuracy during short waits.

This optimization is very effective: log growth is actually 2% lower than reported in Section 6.4, with or without the frame-rate cap. Moreover, the uncapped frame rate is only 3% lower than the rate without the optimization, which shows that the optimization has only a mild impact on game performance.

6.6 Syntactic and semantic check

Alice can audit another player Bob by checking Bob's log against his authenticators (syntactic check) and by replaying Bob's log using a trusted copy of the VM image (semantic check). We expect the syntactic check to be relatively fast, since it is essentially a matter of verifying signatures, whereas the replay involves repeating all the computations that were performed during the original game and should therefore take about as long as the game itself. Our experiments with the log of the server machine from the avmm-rsa768 configuration (which covers 2,216 seconds with 1,987 seconds of actual game play) confirm this. We needed 34.7 seconds to compress the log, 13.2 seconds to decompress it, 6.9 seconds for the syntactic check, and 1,977 seconds for the semantic check (2,031 seconds total). Replay was actually a bit faster because the AVMM skips any time periods in the recording during which the CPU was idle, e.g., before the game was started.

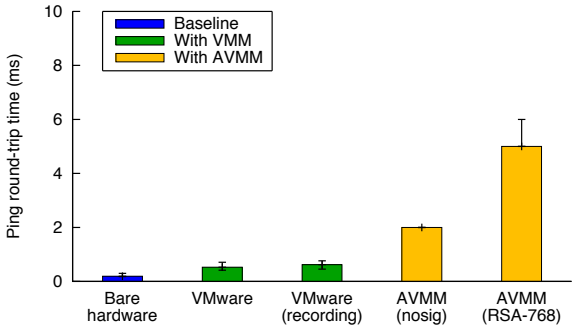


Figure 5: Median ping round-trip times. The error bars show the 5th and the 95th percentile.

Unlike the performance of the actual game, the speed of auditing is not critical because it can be performed at leisure, e.g., in the background while the machine is used for something else.

6.7 Network traffic

The AVMM increases the amount of network traffic for two reasons: First, it adds a cryptographic signature to each packet, and second, it encapsulates all packets in a TCP connection. To quantify this overhead, we measured the raw, IP-level network traffic in the bare-hw configuration and in the avmm-rsa768 configuration. On average, the machine hosting the game sent 22 kbps in bare-hw and 215.5 kbps in avmm-rsa768.

This high relative increase is partly due to the fact that Counterstrike clients send extremely small packets of 50–60 bytes each, at 26 packets/sec, so the AVMM's fixed per-packet overhead (which includes one cryptographic signature for each packet and one for each acknowledgment) has a much higher impact than it would for packets of average Internet packet size. However, in absolute terms, the traffic is still quite low and well within the capabilities of even a slow broadband upstream.

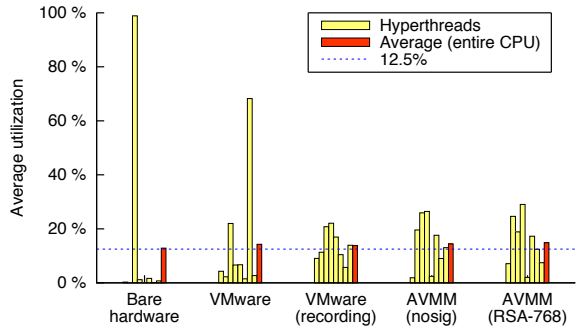


Figure 6: Average CPU utilization in Counterstrike for each of the eight hyperthreads, and for the entire CPU.

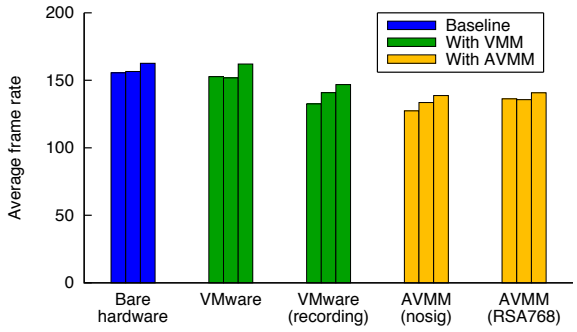


Figure 7: Frame rate in Counterstrike for each of the three machines. The left machine was hosting the game.

6.8 Latency

The AVMM adds some latency to packet transmissions because of the logging and processing of authenticators. To quantify this, we ran an AVM in five different configurations and measured the round-trip time (RTT) of 100 ICMP Echo Request packets. Figure 5 shows the median RTT, as well as the 5th and the 95th percentile. Since our machines are connected to the same switch, the RTT on bare hardware is only 192 μs; adding virtualization increases it to 525 μs, VMware recording to 621 μs, and the daemon to above 2 ms. Enabling 768-bit RSA signatures brings the total RTT to about 5 ms. Recall that both the ping and the pong are acknowledged, so four signatures need to be generated and verified. Since the critical threshold for interactive applications is well above 100 ms [12], 5 ms seem tolerable for games. The overhead could be reduced by using a signing algorithm such as ESIGN [34], which can generate and verify a 2046-bit signature in less than 125 μs.

6.9 CPU utilization

Compared to a Counterstrike game on bare hardware, the AVMM requires additional CPU power for virtualization and for the tamper-evident log. To quantify this overhead, we measured the CPU utilization in five configurations, ranging from bare-hw to avmm-rsa768. To isolate the contribution from the tamper-evident log, we pinned the daemon process to hyperthread 0 (HT 0) in the AVMM experiments and restricted the game to the other hyperthreads except for HT 0's hypertwin, HT 4, which shares a core with HT 0.[2] One of the machines in our experiments runs the Counterstrike server in addition to serving a player. To be conservative, we report numbers for this machine, as it has the highest load.

Figure 6 shows the average utilization for each HT, as well as the average across the entire CPU. The utilization of HT 0 (below 8%) in the AVM experiments shows that the overhead from the tamper-evident log is low. The game is constantly busy rendering frames, but because the Counterstrike rendering engine is single-threaded, it cannot run on more than one HT at a time. The OS/VMM will sometimes schedule it on one HT and sometimes on another, thus we expect an average utilization over the eight HTs of 12.5%, which our results confirm.

[2] Nevertheless, the load on HT 4 is not exactly zero because Linux performs kernel-level IRQ handling on lightly-loaded hyperthreads.

6.10 Frame rate

Since the game is rendering frames as fast as the available CPU cycles allow, a meaningful metric for the CPU overhead is the achieved frame rate, which we consider next. To measure the frame rate, we wrote an AMX Mod X [1] script that increments a counter every time a frame is rendered. We read out this counter at the beginning and at the end of each game, and we divided the difference by the elapsed time. Figure 7 shows our results for each of the three machines. The results vary over time and among players, because the frame rate depends on the complexity of the scene being rendered, and thus on the path taken by each player.

The frame rate on the AVMM is about 13% lower than on bare hardware. The biggest overhead seems to come from enabling recording in VMware Workstation, which causes the average frame rate to drop by about 11%. In absolute terms, the resulting frame rate (137 fps) is still very high; posts in Counterstrike forums generally recommend configuring the game for about 60–80 fps.

To quantify the advantage of running some of the AVMM logic on a different HT, we ran an additional experiment with both Counterstrike and all AVMM threads pinned to the same hyperthread. This reduced the average frame rate by another 11 fps.

6.11 Online auditing

If a game session is long or the stakes are particularly high, players may wish to detect cheaters well before the end of the game. In such cases, players can incre-

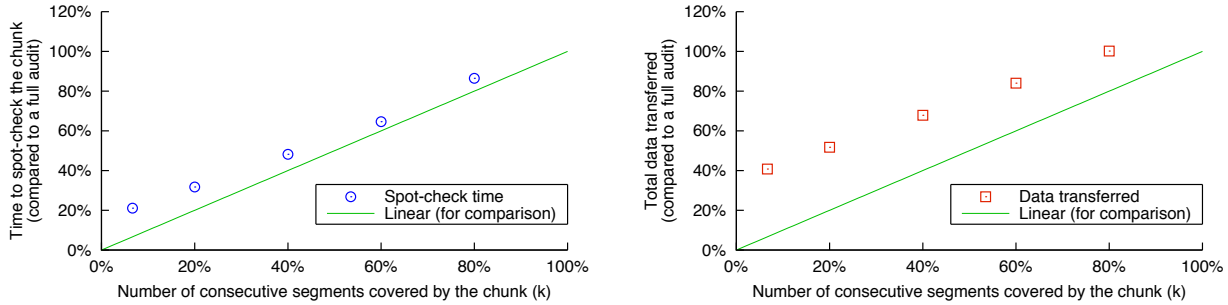


Figure 9: Efficiency of spot checking. The cost of a spot check is roughly proportional to the size of the checked chunk, but there is a fixed cost per chunk for transferring the memory and disk snapshots and for data decompression.

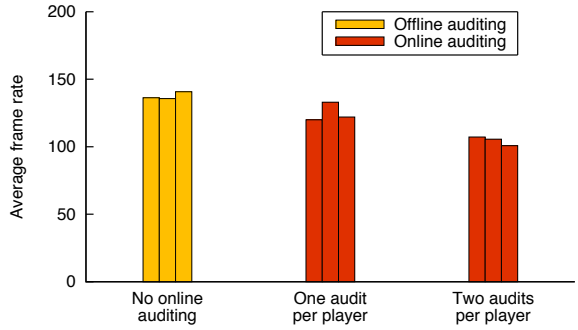


Figure 8: Frame rate for each of the three machines with zero, one, or two online audits per machine.

mentally audit other players' logs while the game is still in progress. In this configuration, which we refer to as *online auditing*, cheating could be detected as soon as the externally visible behavior of the cheater's machine deviates from that of the reference machine.

If a player uses the same machine to concurrently play the game and audit other players, the higher resource consumption can affect game performance. To quantify this effect, we played the game in the avmm-rsa768 configuration with each player auditing zero, one, or two other players on the same machine. As before, we measured the average frame rate experienced by each player.

Figure 8 shows our results. With an increasing number of players audited, the frame rate drops somewhat, from 137 fps with no audits to 104 fps with two audits. However, the drop is less pronounced than expected because the audits can leverage the unused cores. If the number of audits a is increased further, we expect the game performance to eventually degrade with $1/a$.

Since replay is slightly slower than the original execution, auditing falls behind the game by about four seconds per minute of play, even when the audit executes on an otherwise unloaded machine. To ensure quick detection even during very long game sessions, we can compensate by artificially slowing down the original execution. We found that a 5% slowdown was sufficient to allow the auditor to keep up; this reduced the frame rate by up to 7 fps. Note that a certain lag can actually be useful to prevent players from learning each other's current positions or strategies through an audit. In practice, players may want to disallow audits of the current round and/or the most recent moments of game play.

6.12 Spot checking

Online games are not a very interesting use case for spot checking because complete audits are feasible. Therefore, we set up a simple additional experiment that models a client/server system – specifically, a MySQL 5.0.51 server in one AVM and a client running MySQL's `sql-bench` benchmark in another. We ran this experiment for 75 minutes in the avmm-rsa768 configuration, and we recorded a snapshot every five minutes. We found that, on average, our prototype takes 5 seconds to record a snapshot. The incremental disk snapshots are between 1.9 MB and 91 MB, while each memory snapshot occupies about 530 MB. The reason for the latter is that VMware Workstation creates a full dump of the AVM's main memory (512 MB) for each snapshot. This could probably be optimized considerably, e.g., using techniques from Remus [11].

In the following, we refer to the part of the log between two consecutive snapshots as a *segment*, and to k consecutive segments as a *k-chunk*. To quantify the costs of spot checking, we audited all possible k-chunks in our log for $k \in \{1, 3, 5, 9, 12\}$, and measured the amount of data that must be transferred over the network, as well as the time it takes to replay the chunk. However, we excluded k-chunks that start at the beginning of the log; these are atypical because a) they are the only chunks for which no memory or disk snapshots have to be transferred, and b) they have less activity because the MySQL server is not yet running at the beginning. We report averages because the results for chunks with the same value of k never varied by more than 10%.

Figure 9 shows the results, normalized to the cost of a full audit. As expected, the cost grows with the chunk size k; however, there is an additional fixed cost per chunk for transferring the corresponding memory and disk snapshots.

6.13 Summary

Having reported our results, we now revisit our three initial questions. We have demonstrated that our AVMM works out-of-the-box with Counterstrike, a state-of-the-art game, and we have shown that it is effective against real cheats we downloaded from Counterstrike forums on the Internet. AVMs are not free; they affect various metrics such as latency, traffic, or CPU utilization, and they reduce the frame rate by about 13%, compared to the rate achieved on bare hardware. In return for this overhead, players gain the ability to audit other players. Auditing takes time, in some cases as much as the game itself, but it seems time well spent because it either exposes a cheater or clears an innocent player of suspicion. AVMs provide this novel capability by combining two seemingly unrelated technologies, tamper-evident logs and virtualization.

7 Discussion

7.1 Other applications

AVMs are application-independent and can be used in applications other than games.

Distributed systems: AVMs can be used to make any distributed system accountable, simply by executing the software on each node within an AVM. The node software can be arbitrarily complex and available only as a binary system image. Accountability is useful in distributed systems where principals have an interest in monitoring the behavior of other principals' nodes, and where post factum detection is sufficient. Such systems include federated systems where no single entity has complete control or visibility of the entire system, where different parties compete (e.g., in an online game, an auction, or a federated system like the Internet) or where parties are expected to cooperate but lack adequate incentives to do so (e.g., in a peer-to-peer system).

Network traffic accountability: AVMs could also be useful in detecting advanced forms of malware that could escape online detection mechanisms. An AVM, combined with a traffic monitor that records a machine's network communication, can capture the network-observable behavior of a machine, and replay it later with expensive intrusion detection (e.g., taint tracking) in place.

Cloud computing: Another potential application of AVMs is cloud computing. AVMs can enable cloud customers to verify that their software executes in the cloud as expected. AVM are a perfect match for infrastructure-as-a-service (IaaS) clouds that offer customers a virtual machine. However, AVMs in the cloud face additional challenges: auditors cannot easily replay the entire execution for lack of resources; accountable services must be able to interact with non-accountable clients; and, it may not be practical to sign every single packet. The first challenge can be addressed with spot checking (Section 3.5). We plan to address the remaining challenges in future work.

7.2 Using trust to get stronger guarantees

One of the strengths of AVMs is that they can verify the integrity of a remote node's execution without relying on trusted components. However, if trusted components are available, we can obtain additional guarantees. We sketch two possible extensions below.

Secure local input: AVMs cannot detect the hypothetical re-engineered aimbot from Section 5.4 because existing hardware does not authenticate events from local input devices, such as keyboards or mice. Thus, a compromised AVMM can forge or suppress local inputs, and even a correct AVMM cannot know whether a given keystroke was generated by the user or synthesized by another program, or another machine. This limitation can be overcome by adding crypto support to the input devices. For example, keyboards could sign keystroke events before reporting them to the OS, and an auditor could verify that the keystrokes are genuine using the keyboard's public key. Since most peripherals generate input at relatively low rates, the necessary hardware should not be expensive to build.

Trusted AVMM: If we can trust the AVMM that is running on a remote node, we can detect additional classes of cheats and attacks, including certain attacks on confidentiality. For example, a trusted AVMM could establish a secure channel between the AVM and Alice (even if the software in the AVM does not support encryption) and thus prevent Bob's machine from leaking information by secretly communicating with other machines. A trusted AVMM could also prevent wallhacks (see Section 5.3) by controlling outside access to the machine's graphics card. If trusted hardware, such as memory encryption [40] is available on Bob's machine, the AVMM could even prevent Bob from reading information directly from memory. Remote attestation could be used to make sure that a trusted AVMM is indeed running on a remote computer, e.g., using a system like Terra [17].

7.3 Accountability versus privacy

Ideally, an accountability system should disclose to an auditor only the information strictly required to determine that the auditee has met his obligations. By this standard, AVM logs are rather verbose: an AVM records

enough information to replay the execution of the software it is running. This is a price we pay for the generality of AVMs—they can detect a large class of faults in complex software available only in binary form. In practice, the amount of extra information released can be controlled.

Let us consider how the extra information captured in the AVM logs affect Alice and Bob's privacy. The log reveals information about actions of Bob's machine, but only about the execution inside a given AVM, and only to approved auditors. In the web service scenario (Figure 2b), Alice is presumably paying Bob for running her software in an AVM, so she has every right to know about the execution of the software. Similarly, it is not unreasonable to expect players in a game to share information about their game execution. In either case, the auditor cannot observe executions the auditee may be running outside the audited AVM.

Alice and Bob's privacy may be affected when she uses part of the log as evidence to demonstrate a fault on Bob's machine to a third party. The evidence reveals additional information about the AVM, including a snapshot, to that party. Therefore, Alice should release evidence only to third parties that have a legitimate need to know about faults on Bob's machine. To limit the extra information released to third parties, Alice can use the hash tree (Section 4.4) to remove any part of the snapshot that is not necessary to replay the relevant segment.

7.4 Replay for multiprocessors

Our prototype AVMM can assign only a single CPU core for each AVM, because VMware's deterministic replay is limited to uniprocessors. SMP-ReVirt [16] has recently demonstrated that deterministic replay is also possible for multiprocessors, but its cost is substantially higher than the cost of uniprocessor replay. Because replay is a building block for many important applications, such as forensics [15], replication [11], and debugging [25], there is considerable interest in developing more efficient techniques [5, 13, 16, 28, 29]. As more efficient techniques become available, AVMMs can directly benefit from them.

7.5 Bug detection

Recall that AVMs define faults as deviations from the behavior of a reference implementation. If the reference implementation has a bug and this bug is triggered during an execution, it will behave identically during the replay, and thus it will not be classified as a fault. If a bug in the reference implementation permits unauthorized software modification (e.g., a buffer overflow bug), then neither the modification itself nor the behavior of the modified software will be reported as a fault.

Detecting bugs in the reference implementation is outside the fault model AVMs were designed to detect. However, deterministic execution replay provides an opportunity to use sophisticated runtime analysis tools during auditing [10]. In particular, techniques whose runtime costs are too high for deployment in a live system could be used during an off-line replay. Taint tracking, for instance, can reliably detect the unsafe use of data that were received from an untrusted source [33], thus detecting buffer overwrite attacks and other forms of unauthorized software installation. More generally, sophisticated runtime techniques can be used during replay to detect bugs, vulnerabilities and attacks as part of a normal audit.

8 Conclusion

Accountable virtual machines (AVM) allow users to audit software executing on remote machines. An AVM can detect a large and general class of faults, and it produces evidence that can be verified independently by a third party. At the same time, an AVM allows the operator of the remote machine to prove whether his machine is correct. To demonstrate that AVMs are feasible, we have designed and implemented an AVM monitor based on VMware Workstation and used it to detect real cheats in Counterstrike, a popular online multiplayer game. Players can record their game execution in a tamper-evident manner at a modest cost in frame rate. Other players can audit the execution to detect cheats, either after the game has finished or concurrently with the game. The system is able to detect all of 26 existing cheats we examined.

Acknowledgments

We appreciate the detailed and helpful feedback from Jon Howell, the anonymous OSDI reviewers, and our shepherd, Mendel Rosenblum. We would like to thank VMware for making the source code of VMware Workstation available to us under the VMware Academic Program, and our technical contact, Jim Chow, who has been extremely helpful. Finally, we would like to thank our many enthusiastic Counterstrike volunteers.

References

[1] AMX Mod X project. `http://www.amxmodx.org/`.

[2] D. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet protocol (AIP). In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*, Aug. 2008.

[3] K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmerman. Factorization of a 768-bit RSA modulus. `http://eprint.iacr.org/2010/006.pdf`.

[4] K. Argyraki, P. Maniatis, O. Irzak, and S. Shenker. An accountability interface for the Internet. In *Proceedings of the IEEE*

International Conference on Network Protocols (ICNP), Oct. 2007.

[5] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, Oct. 2010.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[7] N. E. Baughman, M. Liberatore, and B. N. Levine. Cheat-proof playout for centralized and peer-to-peer gaming. *IEEE/ACM Transactions on Networking (ToN)*, 15(1):1–13, Feb. 2007.

[8] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[9] C. Chambers, W. Feng, W. Feng, and D. Saha. Mitigating information exposure to cheaters in real-time strategy games. In *Proceedings of the ACM International Workshop on Network and operating systems support for digital audio and video (NOSSDAV)*, June 2005.

[10] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the USENIX Annual Technical Conference*, June 2008.

[11] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2008.

[12] J. Dabrowski and E. V. Munson. Is 100 milliseconds too fast? In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*, Apr. 2001.

[13] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2009.

[14] R. Dingledine, M. J. Freedman, and D. Molnar. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter Accountability. O'Reilly and Associates, 2001.

[15] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2002.

[16] G. W. Dunlap, D. Lucchetti, P. M. Chen, and M. Fetterman. Execution replay for multiprocessor virtual machines. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, Mar. 2008.

[17] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[18] A. Haeberlen. A case for the accountable cloud. In *Proceedings of the ACM SIGOPS International Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, Oct. 2009.

[19] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. Technical Report 2010-3, Max Planck Institute for Software Systems, Sept. 2010.

[20] A. Haeberlen, I. Avramopoulos, J. Rexford, and P. Druschel. NetReview: Detecting when interdomain routing goes wrong. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2009.

[21] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[22] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. Technical Report 2007-3, Max Planck Institute for Software Systems, Oct. 2007.

[23] G. Hoglund. 4.5 million copies of EULA-compliant spyware. `http://www.rootkit.com/blog.php?newsid=358`.

[24] G. Hoglund and G. McGraw. *Exploiting Online Games: Cheating Massively Distributed Systems*. Addison-Wesley, 2007.

[25] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, Apr. 2005.

[26] B. W. Lampson. Computer security in the real world. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec. 2000.

[27] P. Laskowski and J. Chuang. Network monitors and contracting systems: competition and innovation. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*, Sept. 2006.

[28] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira. Offline symbolic analysis for multi-processor execution replay. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2009.

[29] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2010.

[30] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2009.

[31] N. Michalakis, R. Soulé, and R. Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2007.

[32] C. Mönch, G. Grimen, and R. Midtstraum. Protecting online games against cheating. In *Proceedings of the Workshop on Network and Systems Support for Games (NetGames)*, Oct. 2006.

[33] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Annual Network and Distributed Systems Security Symposium (NDSS)*, Feb. 2005.

[34] T. Okamoto. A fast signature scheme based on congruential polynomial operations. *IEEE Transactions on Information Theory*, 36(1):47–53, 1990.

[35] PunkBuster web site. `http://www.evenbalance.com/`.

[36] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

[37] A. Smith. ASUS releases games cheat drivers. `http://www.theregister.co.uk/2001/05/10/asus_releases_games_cheat_drivers/`, May 2001.

[38] Valve Corporation. Valve anti-cheat system (VAC). `https://support.steampowered.com/kb_article.php?ref=7849-RADZ-6869`.

[39] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Annual Workshop on Modeling, Benchmarking, and Simulation (MoBS)*, June 2007.

[40] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin. Improving cost, performance, and security of memory encryption and authentication. *ACM SIGARCH Computer Architecture News*, 34(2):179–190, 2006.

[41] J. Yan and B. Randell. A systematic classification of cheating in online games. In *Proceedings of the Workshop on Network and Systems Support for Games (NetGames)*, Oct. 2005.

[42] S. Yang, A. R. Butt, Y. C. Hu, and S. P. Midkiff. Trust but verify: Monitoring remotely executing programs for progress and correctness. In *Proceedings of the ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2005.

[43] A. R. Yumerefendi and J. S. Chase. Trust but verify: Accountability for Internet services. In *Proceedings of the ACM SIGOPS European Workshop*, Sep 2004.

[44] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. *ACM Transactions on Storage (TOS)*, 3(3):11, Oct. 2007.

Bypassing Races in Live Applications with Execution Filters

Jingyue Wu, Heming Cui, Junfeng Yang
{*jingyue, heming, junfeng*}*@cs.columbia.edu*
Computer Science Department
Columbia University
New York, NY 10027

Abstract

Deployed multithreaded applications contain many races because these applications are difficult to write, test, and debug. Worse, the number of races in deployed applications may drastically increase due to the rise of multicore hardware and the immaturity of current race detectors.

LOOM is a "live-workaround" system designed to quickly and safely bypass application races at runtime. LOOM provides a flexible and safe language for developers to write *execution filters* that explicitly synchronize code. It then uses an *evacuation* algorithm to safely install the filters to live applications to avoid races. It reduces its performance overhead using *hybrid instrumentation* that combines static and dynamic instrumentation.

We evaluated LOOM on nine real races from a diverse set of six applications, including MySQL and Apache. Our results show that (1) LOOM can safely fix all evaluated races in a timely manner, thereby increasing application availability; (2) LOOM incurs little performance overhead; (3) LOOM scales well with the number of application threads; and (4) LOOM is easy to use.

1 Introduction

Deployed multithreaded applications contain many races because these applications are difficult to write, test, and debug. These races include data races, atomicity violations, and order violations [33]. They can cause application crashes and data corruptions. Worse, the number of "deployed races" may drastically increase due to the rise of multicore and the immaturity of race detectors.

Many previous systems can aid race detection (*e.g.,* [31, 32, 37, 47, 54]), replay [9, 18, 28, 36, 43], and diagnosis [42, 49]. However, they do not directly address deployed races. A conventional solution to fixing deployed races is software update, but this method requires application restarts, and is at odds with high availability demand. Live update systems [10, 12, 15, 35, 38, 39, 51] can avoid restarts by adapting conventional patches into *hot patches* and applying them to live systems, but the reliance on conventional patches has two problems.

First, due to the complexity of multithreaded applications, race-fix patches can be *unsafe* and introduce new errors [33]. Safety is crucial to encourage user adoption, yet automatically ensuring safety is difficult because conventional patches are created from general, difficult-to-analyze languages. Thus, previous work [38, 39] had to resort to extensive programmer annotations.

Second, creating a releasable patch from a correct diagnosis can still take time. This delay leaves buggy applications unprotected, compromising reliability and potentially security. This delay can be quite large: we analyzed the Bugzilla records of nine real races and found that this delay can be days, months, or even years. Table 1 shows the detailed results.

Many factors contribute to this delay. At a minimum level, a conventional patch has to go through code review, testing, and other mandatory software development steps before being released, and these steps are all time-consuming. Moreover, though a race may be fixed in many ways (*e.g.,* lock-free flags, fine-grained locks, and coarse-grained locks), developers are often forced to strive for an efficient option. For instance, two of the bugs we analyzed caused long discussions of more than 30 messages, yet both can be fixed by adding a single critical section. Performance pressure is perhaps why many races were *not* fixed by adding locks [33].

This paper presents LOOM, a "live-workaround" system designed to quickly protect applications against races until correct conventional patches are available and the applications can be restarted. It reflects our belief that the true power of live update is its ability to provide immediate workarounds. To use LOOM, developers first compile their application with LOOM. At runtime, to workaround a race, an application developer writes an *execution filter* that synchronizes the application source to filter out racy thread interleavings. This filter is kept separate from the source. Application users can then download the filter and, for immediate protection, install

Race ID	Report	Diagnosis	Fix	Release
Apache-25520	12/15/03	12/18/03	01/17/04	03/19/04
Apache-21287	07/02/03	N/A	12/18/03	03/19/04
Apache-46215	11/14/08	N/A	N/A	N/A
MySQL-169	03/19/03	N/A	03/24/03	06/20/03
MySQL-644	06/12/03	N/A	N/A	05/30/04
MySQL-791	07/04/03	07/04/03	07/14/03	07/22/03
Mozilla-73761	03/28/01	03/28/01	04/09/01	05/07/01
Mozilla-201134	04/07/03	04/07/03	04/16/03	01/08/04
Mozilla-133773	03/27/02	03/27/02	12/01/09	01/21/10

Table 1: *Long delays in race fixing.* We studied the delays in the fix process of nine real races; some of the races were extensively studied [9, 31, 33, 42, 43]. We identify each race by "$Application - \langle Bugzilla\,\# \rangle$." Column **Report** indicates when the race was reported, **Diagnosis** when a developer confirmed the root cause of the race, **Fix** when the final fix was posted, and **Release** when the version of application containing the fix was publicly released. We collected all dates by examining the Bugzilla record of each race. An N/A means that we could not derive the date. The days between diagnosis and fix range from a few days to a month to a few years. For all but two races, the bug reports from the application users contained correct and precise diagnoses. Mozilla-201134 and Mozilla-133773 caused long discussions of more than 30 messages, though both can be fixed by adding a critical region.

it to their application without a restart.

LOOM decouples execution filters from application source to achieve safety and flexibility. Execution filters are safe because LOOM's execution filter language allows only well formed synchronization constraints. For instance, "code region r_1 and r_2 are mutually exclusive." This declarative language is simpler to analyze than a general programing language such as C because LOOM need not reverse-engineer developer intents (*e.g.,* what goes into a critical region) from scattered operations (*e.g.,* `lock()` and `unlock()`).

As temporary workarounds, execution filters are more flexible than conventional patches. One main benefit is that developers can make better performance and reliability tradeoffs during race fixing. For instance, to make two code regions r_1 and r_2 mutually exclusive when they access the same memory object, developers can use critical regions larger than necessary; they can make r_1 and r_2 always mutually exclusive even when accessing different objects; or in extreme cases, they can run r_1 and r_2 in single-threaded mode. This flexibility enables quick workarounds; it can benefit even the applications that do not need live update.

We believe the execution filter idea and the LOOM system as described are worthwhile contributions. To the best of our knowledge, LOOM is the first live-workaround system designed for races. Our additional technical contributions include the techniques we created to address the following two challenges.

A key safety challenge LOOM faces is that even if an execution filter is safe by construction, installing it to a live application can still introduce errors because the application state may be inconsistent with the filter. For instance, if a thread is running inside a code region that an execution filter is trying to protect, a "double-unlock" error could occur. Thus, LOOM must (1) check for inconsistent states and (2) install the filter only in consistent ones. Moreover, LOOM must make the two steps atomic, despite the concurrently running application threads and multiple points of updates. This problem cannot be solved by a common safety heuristic called *function quiescence* [2, 13, 21, 39]. We thus create a new algorithm termed *evacuation* to solve this problem by proactively quiescing an arbitrary set of code regions given at runtime. We believe this algorithm can also benefit other live update systems.

A key performance challenge LOOM faces is to maintain negligible performance overhead during an application's normal operations to encourage adoption. The main runtime overhead comes from the engine used to live-update an application binary. Although LOOM can use general-purpose binary instrumentation tools such as Pin, the overhead of these tools (up to 199% [34] and 1065.39% in our experiments) makes them less suitable as options for LOOM. We thus create a *hybrid instrumentation* engine to reduce overhead. It statically transforms an application to include a "hot backup", which can then be updated arbitrarily by execution filters at runtime.

We implemented LOOM on Linux. It runs in user space and requires no modifications to the applications or the OS, simplifying deployment. It does not rely on non-portable OS features (*e.g.,* `SIGSTOP` to pause applications, which is not supported properly on Windows). LOOM's static transformation is a plugin to the LLVM compiler [3], requiring no changes to the compiler either.

We evaluated LOOM on nine real races from a diverse set of six applications: two server applications, MySQL and Apache; one desktop application PBZip2 (a parallel compression tool); and implementations of three scientific algorithms in SPLASH2 [7]. Our results show that

1. LOOM is effective. It can flexibly and safely fix all races we have studied. It does not degrade application availability when installing execution filters. Its evacuation algorithm can install a fix within a second even under heavy workload, whereas a live update approach using function quiescence cannot install the fix in an hour, the time limit of our experiment.
2. LOOM is fast. LOOM has negligible performance overhead and in some cases even speeds up the applications. The one exception is MySQL. Running MySQL with LOOM alone increases response time by 4.11% and degrades throughput by 3.76%.
3. LOOM is scalable. Experiments on a 48-core ma-

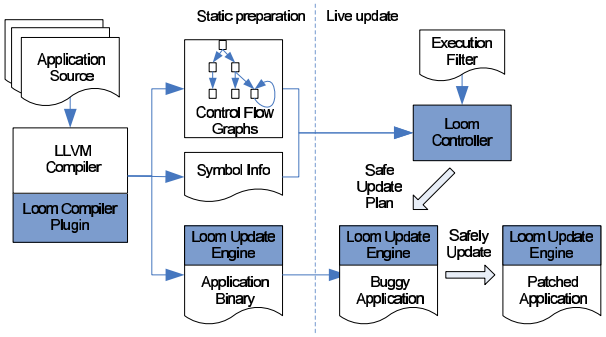


Figure 1: LOOM *overview.* Its components are shaded.

chine show that LOOM scales well as the number of application threads increases.

4. LOOM is easy to use. Execution filters are concise, safe, and flexible (able to fix all races studied, often in more than one way).

This paper is organized as follows. We first give an overview of LOOM (§2). We then describe LOOM's execution filter language (§3), the evacuation algorithm (§4), and the hybrid instrumentation engine (§5). We then present our experimental results (§6). We finally discuss related work (§7) and conclude (§8).

2 Overview

Figure 1 presents an overview of LOOM. To use LOOM for live update, developers first statically transform their applications with LOOM's compiler plugin. This plugin injects a copy of LOOM's update engine into the application binary; it also collects the application's control flow graphs (CFG) and symbol information on behalf of the live update engine.

LOOM's compiler plugin runs within the LLVM compiler [3]. We choose LLVM for its compatibility with GCC and its easy-to-analyze intermediate representation (IR). However, LOOM's algorithms are general and can be ported to other compilers such as GCC. Indeed, for clarity we will present all our algorithms at the source level (instead of the LLVM IR level).

To fix a race, application developers write an execution filter in LOOM's filter language and distribute the filter to application users. A user can then install the filter to immediately protect their application by running

```
% loomctl add <pid> <filter-file>
```

Here `loomctl` is a user-space program called the LOOM controller that interacts with users and initiates live update sessions, `pid` denotes the process ID of a buggy application instance, and `filter-file` is a file containing the execution filter. Under the hood, this controller compiles the execution filter down to a safe update plan using the CFGs and symbol information collected by the compiler plugin. This update plan includes three parts: (1) synchronization operations to enforce the constraints described in the filter and where, in the application, to add the operations; (2) safety preconditions that must hold for installing the filter; and (3) sanity checking code to detect potential errors in the filter itself. The controller sends the update plan to the update engine running as a thread inside the application's address space, which then monitors the runtime states of the application and carries out the update plan only when all the safety preconditions are satisfied.

If LOOM detects a problem with a filter through one of its sanity checks, it can automatically remove the problematic filter. It again waits for all the safety preconditions to hold before removing the filter.

Users can also remove a filter manually, if for example, the race that the filter intends to fix turns out to be benign. They do so by running

```
% loomctl ls <pid>
% loomctl remove <pid> <filter-id>
```

The first command "`loomctl ls`" returns a list of installed filter IDs within process `pid`. The second command "`loomctl remove`" removes filter `filter-id` from process `pid`.

Users can replace an installed filter with a new filter, if for example the new filter fixes the same race but has less performance overhead. Users do so by running

```
% loomctl replace <pid> <old-id> <new-file>
```

where `old-id` is the ID of the installed filter, and `new-file` is a file containing the new filter. LOOM ensures that the removal of the old filter and the installation of the new filter are atomic, so that the application is always protected from the given race.

2.1 Usage Scenarios

LOOM enables users to explicitly describe their synchronization intents and orchestrate thread interleavings of live applications accordingly. Using this mechanism, we envision a variety of strategies users can use to fix races.

Live update At the most basic level, users can translate some conventional patches into execution filters, and use LOOM to install them to live applications.

Temporary workaround Before a permanent fix (*i.e.,* a correct source patch) is out, users can create an execution filter as a crude, temporary fix to a race, to provide immediate protection to highly critical applications.

Preventive fix When a *potential* race is reported (*e.g.,* by automated race detection tools or users of the application), users can immediately install a filter to prevent the race suspect. Later, when developers deem this report false or benign, users can simply remove the filter.

Cooperative fix Users can share filters with each other. This strategy enjoys the same benefits as other cooperative protection schemes [17, 26, 44, 50]. One advantage of LOOM over some of these systems is that it automatically verifies filter safety, thus potentially reducing the need to trust other users.

Site-specific fix Different sites have different workloads. An execution filter too expensive for one site may be fine for another. The flexibility of execution filters allows each site to choose what specific filters to install.

Fix without live update For applications that do not need live update, users can still use LOOM to create quick workarounds, improving reliability.

Besides fixing races, LOOM can be used for the opposite: demonstrating a race by forcing a racy thread interleaving. Compared to previous race diagnosis tools that handle a fixed set of race patterns [25, 41, 42, 49], LOOM's advantage is to allow developers to construct potentially complex "concurrency" testcases.

Although LOOM can also avoid deadlocks by avoiding deadlock-inducing thread interleavings, it is less suitable for this purpose than existing tools (*e.g.,* Dimmunix [26]). To avoid races, LOOM's update engine can add synchronizations to arbitrary program locations. This engine is overkill for avoiding deadlocks: intercepting lock operations (*e.g.,* via `LD_PRELOAD`) is often enough.

2.2 Limitations

LOOM is explicitly designed to work around (broadly defined) races because they are some of the most difficult bugs to fix and this focus simplifies LOOM's execution filter language and safety analysis. LOOM is not intended for other classes of errors. Nonetheless, we believe the idea of high-level and easy-to-verify fixes can be generalized to many other classes of errors.

LOOM does not attempt to fix *occurred* races. That is, if a race has caused bad effects (*e.g.,* corrupted data), LOOM does not attempt to reverse the effects (*e.g.,* recover the data). It is conceivable to allow developers to provide a general function that LOOM runs to recover occurred races before installing a filter. Although this feature is simple to implement, it makes safety analysis infeasible. We thus rejected this feature.

Safety in LOOM terms means that an execution filter and its installation/removal processes introduce no new correctness errors to the application. However, similar to other safe error recovery [46] or avoidance [26, 52] tools, LOOM runs with the application and perturbs timing, thus it may expose some existing application races because it makes some thread interleavings more likely to occur. Moreover, execution filters synchronize code, and may introduce deadlocks and performance problems. LOOM can recover from filter-introduced deadlocks (§3.3) using timeouts, but currently does not deal with performance problems.

At an implementation level, LOOM currently supports a fixed set of synchronization constraint types. Although adding new types of constraints is easy, we have found the existing constraint types sufficient to fix all races evaluated. Another issue is that LOOM uses debugging symbol information in its analysis, which can be inaccurate due to compiler optimization. This inaccuracy has not been a problem for the races in our evaluation because LOOM keeps an unoptimized version of each basic block for live update (§5).

3 Execution Filter Language

LOOM's execution filter language allows developers to explicitly declare their synchronization intents on code. This declarative approach has several benefits. First, it frees developers from the low-level details of synchronization, increasing race fixing productivity. Second, it also simplifies LOOM's safety analysis because LOOM does not have to reverse-engineer developer intents (*e.g.,* what goes into a critical section) from low-level synchronization operations (*e.g.,* scattered `lock()` and `unlock()`), which can be difficult and error-prone. Lastly, LOOM can easily insert error-checking code for safety when it compiles a filter down to low-level synchronization operations.

3.1 Example Races and Execution Filters

In this section, we present two real races and the execution filters to fix them to demonstrate LOOM's execution filter language and its flexibility.

The first race is in MySQL (Bugzilla # 791), which causes the MySQL on-disk transaction log to miss records. Figure 2 shows the race. The code on the left (function `new_file()`) rotates MySQL's transaction log file by closing the current log file and opening a new one; it is called when the transaction log has to be flushed. The code on the right is used by MySQL to append a record to the transaction log. It uses *double-checked locking* and writes to the log only when the log is open. The race occurs if the racy `is_open()` (T2, line 3) catches a closed log when thread T1 is between the `close()` (T1, line 5) and the `open()` (T1, line 6).

Although a straightforward fix to the race exists, performance demands likely forced developers to give up the fix and choose a more complex one instead. The straightforward fix should just remove the racy check (T2, line 3). Unfortunately, this fix creates unnecessary overhead if MySQL is configured to skip logging for speed; this overhead can increase MySQL's response time by more than 10% as observed in our experiments. The concern to this overhead likely forced MySQL developers to use a more involved fix, which adds a new flag field to MySQL's transaction log and modifies the

```
1: // log.cc. thread T1
2: void MYSQL_LOG::new_file(){
3:     lock(&LOCK_log);
4:     ...
5:     close(); // log is closed
6:     open(...);
7:     ...
8:     unlock(&LOCK_log);
9: }
```

```
1: // sql_insert.cc. thread T2
2: // [race] may return false
3: if (mysql_bin_log.is_open()){
4:     lock(&LOCK_log);
5:     if (mysql_bin_log.is_open()){
6:         ... // write to log
7:     }
8:     unlock(&LOCK_log);
9: }
```

Figure 2: *A real MySQL race, slightly modified for clarity.*

```
// Execution filter 1: unilateral exclusion
{log.cc:5, log.cc:6} <> *

// Execution filter 2: mutual exclusion of code
{log.cc:5, log.cc:6} <> MYSQL_LOG::is_open

// Execution filter 3: mutual exclusion of code and data
{log.cc:5 (this), log.cc:6 (this)} <> MYSQL_LOG::is_open(this)
```

Figure 3: *Execution filters for the MySQL race in Figure 2.*

`close()` function to distinguish a regular `close()` call and one for reopening the log.

In contrast, LOOM allows developers to create temporary workarounds with flexible performance and reliability tradeoffs. These temporary fixes can protect the application until developers create a correct and efficient fix at the source level. Figure 3 shows several execution filters that can fix this race. Execution filter 1 in the figure is the most conservative fix: it makes the code region between T1, line 5 and T1, line 6 atomic against all code regions, so that when a thread executes this region, all other threads must pause. We call such a synchronization constraint *unilateral exclusion* in contrast to mutual exclusion that requires participating threads agree on the same lock.[1] Here operator "<>" expresses mutual exclusion constraints, its first operand "{`log.cc:5, log.cc:6`}" specifies a code region to protect, and its second operand "$\star$" represents all code regions. This "expensive" fix incurs only 0.48% overhead (§6.1) because the log rotation code rarely executes.

Execution filter 2 reduces overhead by refining the "$\star$" operand to a specific code region, function `MYSQL_LOG::is_open()`. This filter makes the two code regions mutually exclusive, regardless of what memory locations they access. Execution filter 3 further improves performance by specifying the memory location accessed by each code region.

The second race causes PBZip2 to crash due to a use-after-free error. Figure 4 shows the race. The crash occurs when `fifo` is dereferenced (line 10) after it is freed (line 5). The reason is that the `main()` thread does not wait for the `decompress()` threads to finish. To fix this race, developers can use the filter in Figure 5, which constrains line 10 to run for `numCPU` times before line 5.

[1]Note that unilateral exclusion differs (subtly) from single-threaded execution: unilateral exclusion allows no context switches.

```
// pbzip2.cpp. thread T1
1: main() {
2:     for(i=0;i<numCPU;i++)
3:         pthread_create(...,
4:             decompress, fifo);
5:     queueDelete(fifo);
6: }
```

```
// pbzip2.cpp. thread T2
7 : void *decompress(void *q){
8 :     queue *fifo = (queue *)q;
9 :     ...
10:     pthread_mutex_lock(fifo->mut);
11:     ...
12: }
```

Figure 4: *A real PBZip2 race, simplified for clarity.*

pbzip2.cpp:10 {numCPU} > pbzip2.cpp:5

Figure 5: *Execution filter for the PBZip2 race in Figure 4.*

3.2 Syntax and Semantics

Table 2 summarizes the main syntax and semantics of LOOM's execution filter language. This language allows developers to express synchronization constraints on *events* and *regions*. An event in the simplest form is "$file : line$," which represents a dynamic instance of a static program statement, identified by file name and line number. An event can have an additional "$(expr)$" component and an "$\{n\}$" component, where $expr$ and n refer to valid expressions with no function calls or dereferences. The $expr$ expression distinguishes different dynamic instances of program statements and LOOM synchronizes the events only with matching $expr$ values. The n expression specifies the number of occurrences of an event and is used in execution order constraints. A *region* represents a dynamic instance of a static code region, identified by a set of entry and exist events or an application function. A region representing a function call can have an additional "$(args)$" component to distinguish different calls to the same function.

LOOM currently supports three types of synchronization constraints (the bottom three rows in Table 2). Although adding new constraint types is easy, we have found existing ones enough to fix all races evaluated. An execution order constraint as shown in the table makes event e_1 happen before e_2, e_2 before e_3, and so forth. A mutual exclusion constraint as shown makes every pair of code regions r_i and r_j mutually exclusive with each other. A unilateral exclusion constraint conceptually makes the execution of a code region single-threaded.

3.3 Language Implementation

LOOM implements the execution filter language using locks and semaphores. Given an execution order constraint $e_i > e_{i+1}$, LOOM inserts a semaphore `up()` operation at e_i and a `down()` operation at e_{i+1}. LOOM implements a mutual exclusion constraint by inserting `lock()` at region entries and `unlock()` at region exits. LOOM implements a unilateral exclusion constraint reusing the evacuation mechanism (§4), which can pause threads at safe locations and resume them later.

Constructs	Syntax
Event (short as e)	$file : line$ $file : line\,(expr)$ $e\{n\}$, n is # of occurrence
Region (short as r)	$\{e_1, ..., e_i; e_{i+1}, ..., e_n\}$ $func\,(args)$
Execution Order	$e_1 > e_2 > ... > e_n$
Mutual Exclusion	$r_1 <> r_2 <> ... <> r_n$
Unilateral Exclusion	$r <> *$

Table 2: *Execution filter language summary.*

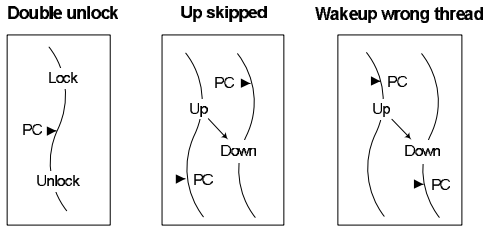


Figure 6: *Unsafe program states for installing filters.*

LOOM creates the needed locks and semaphores on demand. The first time a lock or semaphore is referenced by one of the inserted synchronization operations, LOOM creates this synchronization object based on the ID of the filter, the ID of the constraint, and the value of $expr$ if present. It initializes a lock to an unlocked state and a semaphore to 0. It then inserts this object into a hash table for future references. To limit the size of this table, LOOM garbage-collects these synchronization objects. Freeing a synchronization object is safe as long as it is unlocked (for locks) or has a counter of 0 (for semaphores). If this object is referenced later, LOOM simply re-creates it. The default size of this table is 256 and LOOM never needed to garbage-collect synchronization objects in our experiments.

The `up()` and `down()` operations LOOM inserts behave slightly differently than standard semaphore operations when n, the number of occurrences, is specified. Given $e1\{n_1\} > e2\{n_2\}$, `up()` conceptually increases the semaphore counter by $\frac{1}{n_1}$ and `down()` decreases it by $\frac{1}{n_2}$. Our implementation uses integers instead of floats. LOOM stores the value of n the first time the corresponding event runs and ignores future changes of n.

LOOM computes the values of $expr$ and n using debugging symbol information. We currently allow $expr$ and n to be the following expressions: `a` (constant or primitive variable), `a+b`, `&a`, `&a[i]`, `&a->f`, or any recursive combinations of these expressions. For safety, we do not allow function calls or dereferences. These expressions are sufficient for writing the execution filters in our evaluation.

We implemented this feature using the DWARF library and the `parse_exp_1()` function in GDB. Specifically, we use `parse_exp_1()` to parse the $expr$ or n component into an expression tree, then compile this tree into low level instructions by querying the DWARF library. Note this compilation step is done inside the LOOM controller, so that the live update engine does not have to pay this overhead.

LOOM implements three mechanisms for safety. First, by keying synchronization objects based on filter and constraint IDs, it uses a disjoint set of synchronization objects for different execution filters and constraints, avoiding interference among them. Second, LOOM inserts additional checking code when it generates the update plan. For example, given a code region c in a mutual exclusion constraint, LOOM checks for errors such as c's `unlock()` releasing a lock not acquired by c's `lock()`. Lastly, LOOM checks for filter-induced deadlocks to guard against buggy filters. If a buggy filter introduces a deadlock, one of its synchronization operations must be involved in the wait cycle. LOOM detects such deadlocks using timeouts, and automatically removes the offending filter.

4 Avoiding Unsafe Application States

Figure 6 shows three unsafe scenarios LOOM must handle. For a mutual exclusion constraint that turns code regions into critical sections, LOOM must ensure that no thread is executing within the code regions when installing the filter to avoid "double-unlock" errors. Similarly, for an execution order constraint $e_1 > e_2$, LOOM must ensure either of the following two conditions when installing the filter: (1) both e_1 and e_2 have occurred or (2) neither has occurred; otherwise the `up()` LOOM inserts at e_1 may get skipped or wake up a wrong thread.

Note that a naïve approach is to simply ignore an `unlock()` if the corresponding lock is already unlocked, but this approach does not work with execution order constraints. Moreover, it mixes unsafe program states with buggy filters, and may reject correct filters simply because it tries to install the filters at unsafe program states.

A common safety heuristic called *function quiescence* [2, 13, 21, 39] cannot address this unsafe state problem. This technique updates a function only when no stack frame of this function is active in any call stack of the application. Unfortunately, though this technique can ensure safety for many live updates, it is insufficient for execution filters because their synchronization constraints may affect multiple functions.

We demonstrate this point using a race example. Figure 7 shows the worker thread code of a contrived database. Function `process_client()` is the main thread function. It takes a client socket as input and repeatedly processes requests from the socket. For each request, function `process_client()` opens the corresponding database table by calling `open_table()`, serves the request, and closes the table by calling

```
1 :  // database worker thread
2 :  void handle_client(int fd) {
3 :    for(;;) {
4 :      struct client_req req;
5 :      int ret = recv(fd, &req, ...);
6 :      if(ret <= 0) break;
7 :      open_table(req.table_id);
8 :      ... // do real work
9 :      close_table(req.table_id);
10:    }
11:  }
12:  void open_table(int table_id) {
13:      // fix: acquire table lock
14:      ... // actual code to open table
15:  }
16:  void close_table(int table_id) {
17:      ... // actual code to close table
18:      // fix: release table lock
19:  }
```

Figure 7: *A contrived race.*

`close_table()`. The race in Figure 7 occurs when multiple clients concurrently access the same table.

To fix this race, an execution filter can add a lock acquisition at line 13 in `open_table()` and a lock release at line 18 in `close_table()`. To safely install this filter, however, the quiescence of `open_table()` and `close_table()` is not enough, because a thread may still be running at line 8 and cause a double-unlock error. An alternative fix is to add the lock acquisition and release in function `handle_client()`, but this function hardly quiesces because of the busy loop (line 3-10) and the blocking call `recv()`.

LOOM solves the unsafe state program using an algorithm termed *evacuation* that can proactively quiesce arbitrary code regions. From a high level, this algorithm takes a filter and computes a set of unsafe program locations that may interfere with the filter. It does so conservatively to avoid marking an unsafe location as safe. Then, it "evacuates" threads out of the unsafe locations and blocks them at safe program location. After that, it installs the filter and resumes the threads.

4.1 Computing Unsafe Program Locations

LOOM uses slightly different methods to compute the unsafe program locations for mutual exclusion and for execution order constraints. To compute unsafe program locations for mutual exclusion constraints, LOOM performs a static reachability analysis on the *interprocedural control flow graph (ICFG)* of an application. An ICFG connects each function's control flow graphs by following function calls and returns. Figure 8a shows the ICFG for the code in Figure 7. We say statement s_1 *reaches* s_2 or $reachable(s_1, s_2)$ if there is a path from s_1 to s_2 on the ICFG. For example, the statement at line 13 reaches the statement at line 8 in Figure 7.

Given an execution filter f with mutual exclusion constraint r_1 <> r_2 <> ... <> r_n, LOOM includes any statement s potentially inside one of the regions in $unsafe(f)$, the set of unsafe program locations for filter f. Specifically, $unsafe(f)$ is the set of statements s such that $\{reachable(r_i.entries, s) \wedge reachable(s, r_i.exits)\}$ for $i \in [1, n]$, where $r_i.entries$ are the entry statements to region r_i and $r_i.exits$ are the exit statements.

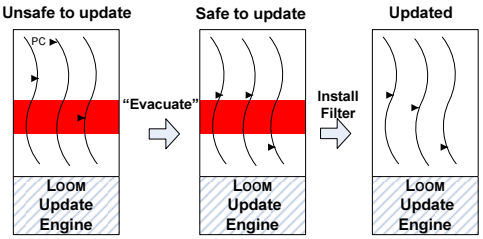


Figure 9: *Evacuation.* Curved lines represent application threads, solid triangles (in black) represents the threads' program counters (PC), and solid stripes (in red) represents an unsafe code region.

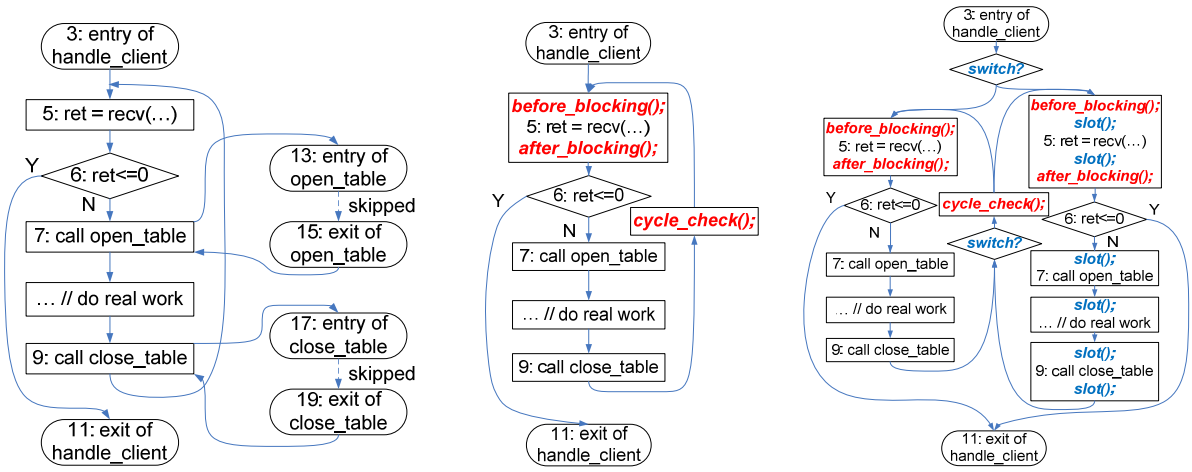
LOOM computes unsafe program locations for an execution order constraint by first deriving code regions from the constraint, then reusing the method for mutual exclusion to compute unsafe program locations. Specifically, given $e_1 > e_2 > ... > e_n$, LOOM first computes a *dominator* statement s_d such that s_d dominates all e_i (*i.e.,* s_d is on every path from the program start to e_i); it then computes $unsafe(f)$ as the set of statements inside each $\{s_d; e_i\}$ region.

Since e_i may be in different threads, LOOM augments the ICFG of an application into *thread interprocedural control flow graph (TICFG)* by adding edges for thread creation and thread join statements. Currently our analysis constructs the TICFG by treating each `pthread_create(func)` statement as a function call to `func()`: it adds an edge from the statement to the entry of `func()` and a thread join edge from the exit of `func()` to the statement.

4.2 Controlling Application Threads

LOOM needs to control application threads to pause and resume them. It does so using a read-write lock called the update lock. To live update an application, LOOM grabs this lock in write mode, performs the update, and releases this lock. To control application threads, LOOM's compiler plugin instruments the application so that the application threads hold this lock in read mode in normal operation and check for an update once in a while by releasing and re-grabbing this lock.

LOOM carefully places update-checks inside an application to reduce the overhead and ensure a timely update. Figure 8b shows the placement of these update-checks. LOOM needs no update-checks inside straight-line code with no blocking calls because such code can complete



(a) *ICFG of the code in Figure 7.* (b) *After update-checks inserted.* (c) *After basic blocks cloned.*

Figure 8: *Static transformations that* LOOM *does for safe and fast live update.* Subfigure (a) shows the ICFG of the code in Figure 7; (b) shows the resulting CFG of function `process_client()` after the instrumentation to control application threads (§4); (c) shows the final CFG of function `process_client()` after basic block cloning (§5).

quickly. LOOM places one update-check for each cycle in the control flow graph, including loops and recursive function call chains, so that an application thread cycling in one of these cycles can check for an update at least once each iteration. Currently LOOM instruments the backedge of a loop and an arbitrary function entry in a recursive function cycle. LOOM does not instrument every function entry because doing so is costly.

LOOM also instruments an application to release the update lock before a blocking call and re-grab it after the call, so that an application thread blocking on the call does not delay an update. For the example in Figure 7, LOOM can perform the update despite some threads blocking in `recv()`. LOOM instruments only the "leaf-level" blocking calls. That is, if `foo()` calls `bar()` and `bar()` is blocking, LOOM instruments the calls to `bar()`, but not the calls to `foo()`. Currently LOOM conservatively considers calls to external functions (*i.e.,* functions without source), except Math library functions, as blocking to save user annotation effort.

4.3 Pausing at Safe Program Locations

Besides the update lock, LOOM uses additional synchronization variables to ensure that application threads pause at safe locations. LOOM assigns a wait flag for each backedge of a loop and the chosen function entry of a recursive call cycle. To enable/disable pausing at a safe/unsafe location, LOOM sets/clears the corresponding flag. The instrumentation code for each CFG cycle (left of Figure 10) checks for an update only when the corresponding wait flag is set. These wait flags allow application threads at unsafe program locations to run until they reach safe program locations, effectively evacuating the unsafe program locations.

```
// inserted at CFG cycle
void cycle_check() {
  if(wait[stmt_id]) {
    read_unlock(&update);
    while(wait[stmt_id]);
    read_lock(&update);
  }
}
```

```
// inserted before blocking call
void before_blocking() {
  atomic_inc(&counter[callsite_id]);
  read_unlock(&update);
}
// inserted after blocking call
void after_blocking() {
  read_lock(&update);
  atomic_dec(&counter[callsite_id]);
}
```

Figure 10: *Instrumentation to pause application threads.*

Note that the statement "`if(wait[stmt_id])`" in Figure 10 greatly improves LOOM's performance. With this statement, application threads need not always release and re-grab the update lock which can be costly, and hardware cache and branch prediction can effectively hide the overhead of checking these flags. This technique speeds up LOOM significantly (§6) because wait flags are almost always 0 with read accesses.

LOOM cannot use the wait-flag technique to skip a blocking function call because doing so changes the application semantics. Instead, LOOM assigns a counter to each blocking callsite to track how many threads are at the callsites (right of Figure 10). LOOM uses a counter instead of a binary flag because multiple threads may be doing the same call.

Now that LOOM's instrumentation is in place, Figure 11 shows LOOM's evacuation method which runs within LOOM's live update engine. This method first sets the wait flags for safe backedges. It then grabs the update lock in write mode, which pauses all application threads. It then examines the counters of unsafe callsites and if any counter is positive, it releases the update lock and retries, so that the thread blocked at unsafe callsites can

```
volatile int wait[NBACKEDGE] = {0};
volatile int counter[NCALLSITE] = {0};
rwlock_t update;
void evacuate() {
  for each B in safe backedges
    wait[B] = 1; // turn on wait flags
retry:
  write_lock(&update); // pause app threads
  for each C in unsafe callsites
    if(counter[C]) { // threads paused at unsafe callsites
      write_unlock(&update);
      goto retry;
    }
  ... // update
  for each B in safe backedges
    wait[B] = 0; // turn off wait flags
  write_unlock(&update); // resume app threads
}
```

Figure 11: *Pseudo code of the evacuation algorithm.*

wake up and advance to safe locations. Next, it updates the application (§5), clears the wait flags, and releases the update lock.

4.4 Correctness Discussion

We briefly discuss the correctness of our evacuation algorithm in this subsection; for a complete proof, please refer to our technical report [53].

In program analysis terms, our reachability analysis (§4.1) is interprocedural and flow-sensitive. We use a crude pointer analysis to discover thread functions, thread join sites, and function pointer targets. We could have refined our analysis to improve precision, but we find it sufficient to compute unsafe locations for all evaluated races because (1) our analysis is sound and never marks an unsafe location safe and (2) execution filters are quite small and slight imprecision does not matter. In the worst case, if our analysis turns out too imprecise for some filters, the flexibility of LOOM allows developers to easily adjust their filters to pass the safety analysis.

Server programs frequently use thread pools, creating problems for our reachability analysis. Specifically, these servers tend to create a fixed set of threads during initialization, then reuse them for independent requests. If we compute dominators using the creation sites of these threads, we would find that dominators only run during server initialization. Fortunately, we can annotate the reuse of a thread as a special thread creation site, so that our algorithm computes correct dominators. In our experiments, we did not (and need not) annotate any thread reuse.

Our reachability analysis gives correct results despite compiler reordering. In order to pause application threads at safe locations, our reachability analysis returns only the set of unsafe backedges and external callsites. These locations are instrumented by LOOM; this instrumentation acts as barriers and prevents compilers from reordering instructions across them.

```
void slot(int stmt_id) {
  op_list = operations[stmt_id];
  foreach op in op_list
    do op;
}
```

Figure 12: *Slot function.*

The synchronization between the instrumentation in Figure 10 and the evacuation algorithm in Figure 11 is correct under two conditions: (1) read and write to wait flags are atomic and (2) the operations to the update lock contain correct memory barriers that prevent hardware reordering. Currently we implement wait flags using aligned integers; our update lock operations use atomic operations similar to the Linux kernel's `rw_spinlock`. Thus, our evacuation algorithm works correctly on X86 and AMD64 which do not reorder instructions across atomic instructions. We expect our algorithm to work on other commodity hardware that also provides this guarantee. To cope with more relaxed hardware (*e.g.,* , Alpha), we can augment these operations with full barriers.

5 Hybrid Instrumentation

Most previous live update systems update binaries by compiling updated functions and redirecting old functions to the new function binaries using a table or jump instructions. This approach requires source patches to generate the updates, thus it has the limitations described in §1. Moreover, this approach pays the overhead of position independent code (PIC) because application functions must be compiled as PIC for live update. It also suffers the aforementioned function quiescence problem.[2]

Another alternative is to use general-purpose binary instrumentation tools such as vx32 [20], Pin [34] and DynamoRIO [14], but they tend to incur significant runtime overhead just to run their frameworks alone. For example, Pin has been reported to incur 199% overhead [34], and we observed 10 times slowdown on Apache with a CPU-bound workload (§6).

LOOM's hybrid instrumentation engine reduces runtime overhead by combining static and dynamic instrumentation. This engine statically transforms an application's binary to anticipate dynamic updates. The static transformation pre-pads, before each program location, a *slot function* which interprets the updates to this program location at runtime. Figure 12 shows the pseudo code of this function. It iterates though a list of synchronization operations assigned to the current statement and performs each. To update a program location at runtime, LOOM simply modifies the corresponding operation list.

Inserting the slot function at every statement incurs

[2]The function quiescence problem can be addressed by transforming loop bodies into functions [38, 39] but only if the CFGs are reducible [23].

Race ID	Description
MySQL-791	Calls to `close()` and `open()` to flush log file are not atomic. Figure 2 shows the code.
MySQL-169	Table update and log write in `mysql_delete()` are not atomic.
MySQL-644	Calls to `prepare()` and `optimize()` in `mysql_select()` are not atomic.
Apache-21287	Reference count decrement and checking are not atomic.
Apache-25520	Threads write to same log buffer concurrently, resulting in corrupted logs or crashes.
PBZip2	Variable `fifo` is used in one thread after being freed by another. Figure 4 shows the code.
SPLASH2-fft	Variable `finishtime` and `initdonetime` are read before assigned the correct values.
SPLASH2-lu	Variable `rf` is read before assigned the correct value.
SPLASH2-barnes	Variable `tracktime` is read before assigned the correct value.

Table 3: *All races used in evaluation.* We identify races in MySQL and Apache as "$\langle application\ name \rangle - \langle Bugzilla\ \# \rangle$", the only race in PBZip2 "$PBZip2$", and races in SPLASH2 "$SPLASH2 - \langle benchmark\ name \rangle$".

high runtime overhead and hinders compiler optimization. LOOM solves this problem using a basic block cloning idea [29]. LOOM keeps two versions of each basic block in the application binary, an originally compiled version that is optimized, and a hot backup that is unoptimized and padded for live update. To update a basic block at runtime, LOOM simply updates the backup and switches the execution to the backup by flipping a switch flag.

LOOM instruments only function entries and loop backedges to check the switch flags because doing so for each basic block is expensive. Similar to the wait flags in (§4), the switch flags are almost always 0, so that hardware cache and branch predication can effectively hide the overhead of checking them. This technique makes live-update-ready applications run as fast as the original application during normal operations (§6). Figure 8c shows the final results after all LOOM transformations.

Note that the accesses to switch flags are correctly protected by the update lock. An application checks the switch flag when holding the update lock in read mode, and the update engine sets the switch flag when holding the update lock in write mode.

6 Evaluation

We implemented LOOM in Linux. It consists of 4,852 lines of C++ code, with 1,888 lines for the LLVM compiler plugin, 2,349 lines for the live-update engine, and 615 lines for the controller.

We evaluated LOOM on nine real races from a diverse set of applications, ranging from two server applications MySQL [5] and Apache [11], to one desktop application PBZip2 [6], to three scientific applications fft, lu, and barnes in SPLASH2 [7].[3] Table 3 lists all nine races. Our race selection criteria is simple: (1) they are extensively used in previous studies [31, 42, 43] and (2) the application can be compiled by LLVM and the race can be reproduced on our main evaluation machine, a 2.66 GHz Intel quad-core machine with 4 GB memory running 32-bit Linux 2.6.24.

We used the following workloads in our experiments. For MySQL, we used SysBench [8] (advanced transaction workload), which randomly selects, updates, deletes, and inserts database records. For Apache, we used ApacheBench [1], which repeatedly downloads a webpage. Both benchmarks are multithreaded and used by the server developers. We made both SysBench and ApacheBench CPU bound by fitting the database or web contents within memory; we also ran both the client and the server on the same machine, to avoid masking LOOM's overhead with the network overhead. Unless otherwise specified, we ran 16 worker threads for MySQL and Apache because they performed best with 8-16 threads. We ran four worker threads for PBZip2 and SPLASH2 applications because they are CPU-intensive and our evaluation machine has four cores.

We measured throughput (TPUT) and response time (RESP) for server applications and overall execution time for other applications. We report LOOM's relative overhead, the smaller the better. We compiled the applications down to x86 instructions using `llvm-gcc -O2` and LLVM's bitcode compiler `llc`. For all the performance numbers reported, we repeated the experiment 50 times and take the average.

We focus our evaluation on five dimensions:

1. Overhead. Does LOOM incur low overhead?
2. Scalability. Does LOOM scale well as the number of application threads increases?
3. Reliability. Can LOOM be used to fix the races listed in Table 3? What are the performance and reliability tradeoffs of execution filters?
4. Availability. Does LOOM severely degrade application availability when execution filters are installed?
5. Timeliness. Can LOOM install fixes in a timely way?

6.1 Overhead

Figure 13 shows the performance overhead of LOOM during the normal operations of the applications. We also show the overhead of bare Pin for reference. LOOM incurs little overhead for Apache and SPLASH2 benchmarks. It increases MySQL's response time by 4.11% and degrades its throughput by 3.76%. In contrast, Pin incurs higher overhead for all applications evaluated, es-

[3] We include applications that do not need live update for two reasons. First, as discussed in §1, LOOM can provide quick workarounds for these applications as well. Second, we use them to measure LOOM's overhead and scalability.

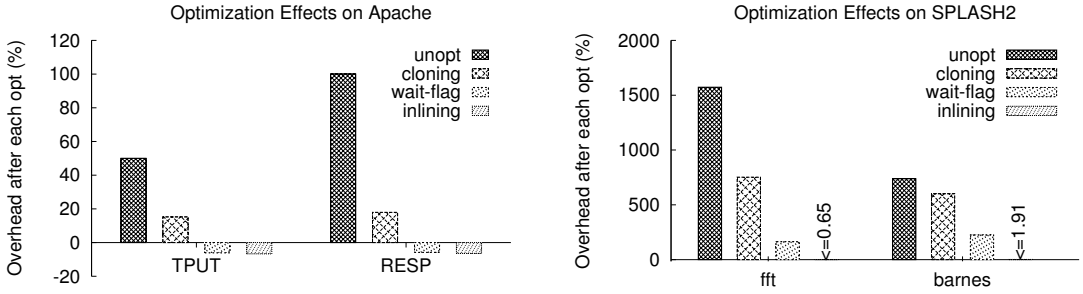


Figure 14: *Effects of* LOOM*'s optimizations.* Label **unopt** represents the versions with no optimizations; **cloning** represents the version with basic block cloning (§5); **wait-flag** represents the version with statement "`if(wait[stmt_id])`" added (§4.2); and **inlining** indicates the version with all LOOM instrumentation inlined into the applications.

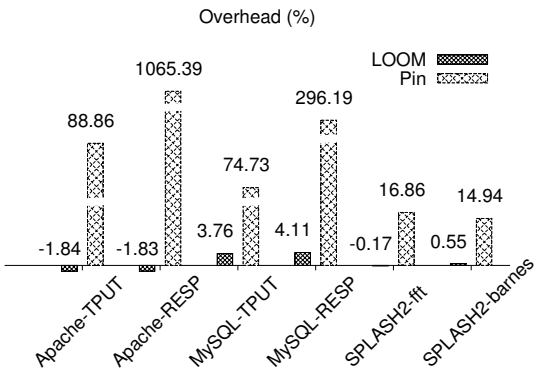


Figure 13: LOOM*'s relative overhead during normal operation.* Smaller numbers are better. We show Pin's overhead for reference. Some Pin bars are broken.

Race ID	Mutual			Unilateral		
	Events	TPUT	RESP	Events	TPUT	RESP
MySQL-169	2	0.14%	0.15%	1	3.28%	3.37%
MySQL-644	4	0.22%	0.20%	4	32.58%	48.34%
MySQL-791	4	0.23%	0.32%	2	0.33%	0.48%
Apache-21287	16	-0.02%	-0.03%	2	54.03%	118.16%
Apache-25520	1	0.52%	0.55%	1	86.04%	637.03%

Table 4: *Execution filter stats for atomicity errors.* Column Events counts the number of events in each filter.

Race ID	Events	Overhead
PBZip2	6	1.26%
SPLASH2-fft	6	0.08%
SPLASH2-lu	2	1.68%
SPLASH2-barnes	2	1.99%

Table 5: *Execution filter stats for order errors.*

pecially for Apache and MySQL.

We also evaluated how the optimizations we do reduce LOOM's overhead. Figure 14 shows the effects of these optimizations. Both cloning and wait-flag are very effective at reducing overhead. Cloning reduces LOOM's response-time overhead on Apache from 100% to 17%. It also reduces LOOM's overhead on fft from 15 times to 8 times. Wait-flag actually makes Apache run faster than the original version. Inlining does not help the servers much, but it does help for SPLASH2 applications.

6.2 Scalability

LOOM synchronizes with application threads via a read-write lock. Thus, one concern is, can LOOM scale well as the number of application threads increases? To evaluate LOOM's scalability, we ran Apache and MySQL with LOOM on a 48-core machine with four 1.9 GHz 12-core AMD CPUs and 64 GB memory running 64-bit Linux 2.6.24. In each experiment, we pinned the benchmark to one CPU and the server to the other three to avoid unnecessary CPU contention between them.

Figure 15 shows LOOM's relative overhead vs. the number of application threads for Apache and MySQL. LOOM scales well with the number of threads. Its relative overhead varies only slightly. Even with 32 server threads, the overhead for Apache is less than 3%, and the overhead for MySQL is less than 12%.

Our initial MySQL overhead was around 16%. We analyzed the execution counts of the LOOM-inserted functions and immediately identified two update-check sites (`cycle_check()` calls) that executed exceedingly many times. These update-check sites are in MySQL functions `ptr_compare_1` and `Field_varstring::val_str`. The first function compares two strings, and the second copies one string to another. Each function has a loop with a few statements and no function calls. Such tight loops cause higher overhead for LOOM, but rarely need to be updated. We thus disabled the update-check sites in these two functions, which reduced the overhead of MySQL down to 12%. This optimization can be easily automated using static or dynamic analysis, which we leave for future work.

6.3 Reliability

LOOM can be used to fix all races evaluated. (We verified this result by manually inspecting the application binary.)

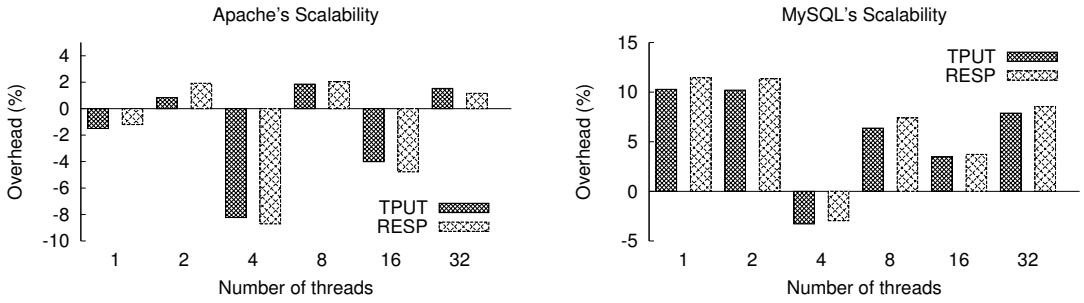


Figure 15: LOOM*'s relative overhead vs. the number of application threads.*

Table 4 shows the statistics for the execution filters that fix atomicity errors. Table 5 shows the statistics for the execution filters that fix order errors.

In all cases, we can fix the race using multiple execution filters, demonstrating the flexibility of LOOM. (The filters for MySQL-791 are shown in Figure 3.) We only show the statistics of one execution filter of each constraint type; other filters of the same type are similar. Our results show that the filters are fairly small, 3.79 events on average and no more than 16 events, demonstrating the ease of use of LOOM. Most filters incur only a small overhead on top of LOOM. Unilateral filters tend to be slightly smaller than mutual exclusion filters, but they can be expensive sometimes. They incur little overhead for two of the MySQL bugs because the code regions protected by the filters rarely run.

These different reliability and performance overheads present an interesting tradeoff to developers. For example, users can choose to install a unilateral filter for immediate protection, then atomically replace it with a faster mutual exclusion filter. Moreover, a user can choose an "expensive" filter as long as their workload is compatible with the filter.

6.4 Availability

We show that LOOM can improve server availability by comparing LOOM to the restart-based software update approach. We restarted a server by running its startup script under `/etc/init.d`. We chose two races, MySQL-791 and Apache-25520, and measured how software updates (conventional or with LOOM) might degrade performance. Note this comparison favors conventional updates because we only compare the installation of the fix, but LOOM also makes it quick to develop fixes. Figure 16 shows the comparison result. Using the restart approach, Apache is unavailable for 4 seconds, and MySQL is unavailable for 2 seconds. Moreover, the restarts also cause Apache and MySQL to lose their internal cache, leading to a ramp-up period after the restart. In contrast, installing an filter using LOOM (at second 5) does not degrade throughput for MySQL and only degrades throughput slightly for Apache.

6.5 Timeliness

The more timely LOOM installs a filter, the quicker the application is protected from the corresponding race. This timeliness is critical for server applications because malicious clients may exploit a known race and launch attacks. In this subsection, we compare how timely LOOM's evacuation algorithm installs an aggressive filter vs. an approach that passively waits for function quiescence. We chose Apache-25520 as the benchmark race. We wrote a simple mutual exclusion filter that fixes the race by making function `ap_buffered_log_writer` a critical region. We then measured the latency from the moment LOOM receives a filter to the moment the filter is installed. We simulated a function quiescence approach by running LOOM without making any `wait_flag` false, so that a thread can pause wherever we insert update-checks. We used the same SysBench and ApacheBench workload. Our results show that LOOM can install the filter within 368 ms. It spends majority of the time waiting for threads to evacuate. In contrast, an approach based on function quiescence fails to install the filter in an hour, our experiment's time limit.

7 Related Work

Live update LOOM differs from previous live update systems [10, 12, 15, 35, 38, 39, 51] in that it is explicitly designed for developers to quickly develop temporary workarounds to races. Moreover, it can automatically ensure the safety of the workarounds. In contrast, previous work focuses only on live update after a source patch is available, thus it does not address the automatic-safety and flexibility problems LOOM addresses.

The live update system closest to LOOM is STUMP [38], which can live-update multithreaded applications written in C. Its prior version Ginseng [39] works with single-threaded C applications. Both STUMP and Ginseng have been shown to be able to apply arbitrary source patches and update applications across major releases. Unlike LOOM, both STUMP and Ginseng require

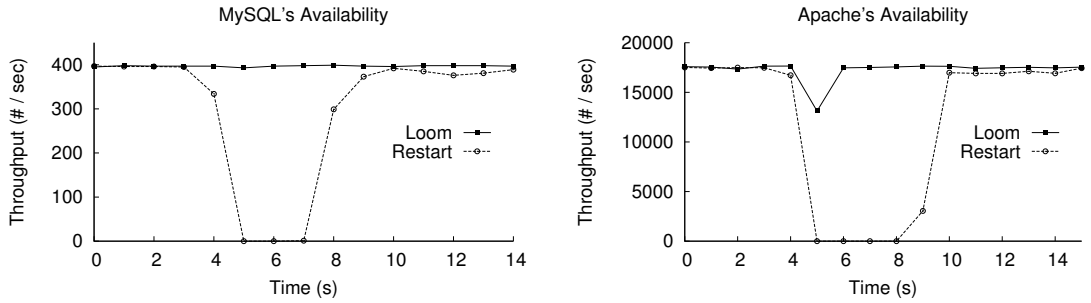


Figure 16: *Throughput degradation for fixing races with* LOOM *vs. with conventional software update.*

source modifications and rely on extensive user annotations for safety because the safety of arbitrary live updates has been proven undecidable [22].

A number of live update systems can update kernels without reboots [12, 15, 35]. The most recent one, Ksplice [12], constructs live updates from object code, and does not require developer efforts to adapt existing source patches. Unlike LOOM, Ksplice uses function quiescence for safety, and is thus prone to the unsafe state problem discussed in §4. Another kernel live update system, DynAMOS [35], requires users to manually construct multiple versions of a function to update non-quiescent functions. This technique is different from basic block cloning (§5): the former is manual and for safety, whereas the later is automatic and for speed.

Error workaround and recovery We compare LOOM to recent error workaround and recovery tools. ClearView [44], ASSURE [50], and Failure-oblivious computing can increase application availability by letting them continue despite errors. Compared to LOOM, these systems are unsafe, and do not directly deal with races. Rx [46] can safely recover from runtime faults using application checkpoints and environment modifications, but it does not fix errors because the same error can re-appear. Vigilante [17] enables hosts to collaboratively contain worms using self-verifiable alerts. By automatically ensuring filter safety, LOOM shares similar benefits.

Two recent systems, Dimmunix [26] and Gadara [52], can fix deadlocks in legacy multithreaded programs. Dimmunix extracts signatures from occurred deadlocks (or starvations) and dynamically avoids them in future executions. Gadara uses control theory to statically transform a program into a deadlock-free program. Both systems have been shown to work on real, large applications. They may possibly be adapted to fix races, albeit at a coarser granularity because these systems control only lock operations.

Kivati [16] automatically detects and prevents atomicity violations for production systems. It reduces performance overhead by cleverly using hardware watch points, but the limited number of watch points on commodity hardware means that Kivati cannot prevent all atomicity violations. Nor does Kivati prevent execution order violations. LOOM can be used to workaround these errors missed by Kivati.

Program instrumentation frameworks Previous work [3, 19, 40] can instrument programs with low runtime overhead, but instrumentation has to be done at compile time. Translation-based dynamic instrumentation frameworks [14, 20, 34] can update programs at runtime but incur high overhead. In particular, vx32 [20] is a novel user-level sandbox that reduces overhead using segmentation hardware; it can be used as an efficient dynamic binary translator. Jump-based instrumentation frameworks [24, 48] have low overhead but automatically ensuring safety for them can be difficult due to low-level issues such as position-dependent code, short instructions, and locations of basic blocks.

One advantage of these instrumentation frameworks over LOOM is that LOOM requires CFGs and symbol information to be distributed to user machines, thus it risks leaking proprietary code information. However, this risk is not a concern for open-source software. Moreover, LOOM only mildly increases this risk because CFGs can often be reconstructed from binaries, and companies such as Microsoft already share symbol information [4].

The advantage of LOOM is that it combines static and dynamic instrumentation, thus allowing arbitrary dynamic updates issued by execution filters with negligible runtime overhead. LOOM borrows basic block cloning from previous work by Liblit *et al.* [29], but their framework is static only. This idea has also been used in other systems (*e.g.,* LIFT [45]).

Other related work Our work was inspired by many observations made by Lu *et al.* [33]. Aspect-oriented programming (AOP) allows developers to "weave" in synchronizations into code [27, 30]. LOOM's execution filter language shares some similarity to AOP, and can be made more expressive by incorporating more aspects. However, to the best of our knowledge, no existing AOP systems were designed to support race fixing at runtime. We

view the large body of race detection and diagnosis work (*e.g.,* [31, 32, 37, 42, 47, 49, 54]) as complimentary to our work and LOOM can be used to fix errors detected and isolated by these tools.

8 Conclusion

We have presented LOOM, a live-workaround system designed to quickly and safely fix application races at runtime. Its flexible language allows developers to write concise execution filters to declare their synchronization intents on code. Its evacuation algorithm automatically ensures the safety of execution filters and their installation/removal processes. It uses hybrid instrumentation to reduce its performance overhead during the normal operations of applications. We have evaluated LOOM on nine real races from a diverse set of applications. Our results show that LOOM is fast, scalable, and easy to use. It can safely fix all evaluated races in a timely manner, thereby increasing application availability.

LOOM demonstrates that live-workaround systems can increase application availability with little performance overhead. In our future work, we plan to extend this idea to other classes of errors (*e.g.,* security vulnerabilities).

Acknowledgement

We thank Cristian Cadar, Jason Nieh, Jinyang Li, Michael Kester, Xiaowei Yang, Vijayan Prabhakaran (our shepherd), and the anonymous reviewers for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We thank Shan Lu for providing many of the races used in our evaluation. We thank Jane-Ellen Long for time management.

This work was supported by the National Science Foundation (NSF) through Contract CNS-1012633 and CNS-0905246 and the United States Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024 and FA8750-10-2-0253. Opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government.

References

[1] ab - Apache HTTP server benchmarking tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[2] The K42 Project. `http://www.research.ibm.com/K42/`.

[3] The LLVM Compiler Framework. `http://llvm.org`.

[4] Download windows symbol packages. `http://www.microsoft.com/whdc/devtools/debugging/debugstart.mspx`.

[5] MySQL Database. `http://www.mysql.com/`.

[6] Parallel BZIP2 (PBZIP2). `http://compression.ca/pbzip2/`.

[7] Stanford Parallel Applications for Shared Memory (SPLASH). `http://www-flash.stanford.edu/apps/SPLASH/`.

[8] SysBench: a system performance benchmark. `http://sysbench.sourceforge.net`.

[9] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 193–206, 2009.

[10] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: online patches and updates for security. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[11] Apache Web Server. `http://www.apache.org`.

[12] J. Arnold and F. M. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems (EUROSYS '09)*, pages 187–198, Apr. 2009.

[13] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 32–32, 2005.

[14] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, 2004. Supervisor-Amarasinghe, Saman.

[15] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*, pages 35–44, 2006.

[16] L. Chew and D. Lie. Kivati: fast detection and prevention of atomicity violations. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 307–320, 2010.

[17] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 133–147, 2005.

[18] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE '08)*, pages 121–130, 2008.

[19] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, Sept. 2000.

[20] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, pages 293–306, 2008.

[21] S. Gilmore and C. Walton. Dynamic ML without dynamic types. Technical report, Lab. for the Foundations of Computer Science, University of Edinburgh, 1997.

[22] D. Gupta, P. Jalote, and G. Barua. A formal framework for online software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

[23] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, 1974.

[24] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *In Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, 1998.

[25] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, pages 110–120, June 2009.

[26] H. Jula, D. Tralamazza, Z. Cristian, and C. George. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 295–308, Dec. 2008.

[27] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, 1997.

[28] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the 2010 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2010.

[29] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 141–154, 2003.

[30] D. Lohmann, W. Hofer, W. Schrder-Preikschat, J. Streicher, and O. Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '09)*, 2009.

[31] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Twelfth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '06)*, pages 37–48, Oct. 2006.

[32] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *SIGOPS Oper. Syst. Rev.*, 41(6):103–116, 2007.

[33] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339, 2008.

[34] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 190–200, 2005.

[35] K. Makris and K. Ryu. Dynamic and adaptive updates of nonquiescent subsystems in commodity operating system kernels. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, page 340, 2007.

[36] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 73–84, 2009.

[37] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 267–280, Dec. 2008.

[38] I. Neamtiu and M. Hicks. Safe and timely dynamic updates for multi-threaded programs. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, pages 13–24, June 2009.

[39] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. pages 72–83, June 2006.

[40] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of Conference on Compilier Construction*, March 2002.

[41] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*, pages 135–145, Nov. 2008.

[42] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 25–36, Mar. 2009.

[43] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 177–192, 2009.

[44] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 87–102, 2009.

[45] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, 2006.

[46] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Trans. Comput. Syst.*, 25(3):7, 2007.

[47] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, pages 391–411, Nov. 1997.

[48] M. Schulz, D. Ahn, A. Bernat, B. R. de Supinski, S. Y. Ko, G. Lee, and B. Rountree. Scalable dynamic binary instrumentation for blue gene/l. *SIGARCH Comput. Archit. News*, 33(5): 9–14, 2005.

[49] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI '08)*, pages 11–21, June 2008.

[50] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. ASSURE: automatic software self-healing using rescue points. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 37–48, 2009.

[51] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a vm-centric approach. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, pages 1–12, 2009.

[52] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 281–294, Dec. 2008.

[53] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. Technical report, Columbia University.

[54] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234, 2005.

Effective Data-Race Detection for the Kernel

John Erickson, Madanlal Musuvathi,
Sebastian Burckhardt, Kirk Olynyk
Microsoft Research
{jerick, madanm, sburckha, kirko}@microsoft.com

Abstract

Data races are an important class of concurrency errors where two threads erroneously access a shared memory location without appropriate synchronization. This paper presents *DataCollider,* a lightweight and effective technique for dynamically detecting data races in kernel modules. Unlike existing data-race detection techniques, DataCollider is oblivious to the synchronization protocols (such as locking disciplines) the program uses to protect shared memory accesses. This is particularly important for low-level kernel code that uses a myriad of complex architecture/device specific synchronization mechanisms. To reduce the runtime overhead, DataCollider randomly samples a small percentage of memory accesses as candidates for data-race detection. The key novelty of DataCollider is that it uses breakpoint facilities already supported by many hardware architectures to achieve negligible runtime overheads. We have implemented DataCollider for the Windows 7 kernel and have found 25 confirmed erroneous data races of which 12 have already been fixed.

1. Introduction

Concurrent systems are hard to design, arguably because of the difficulties of finding and fixing concurrency errors. Data races are an important class of concurrency errors, where the program fails to use proper synchronization when accessing shared data. The effects of an erroneous data race can range from immediate program crashes to silent lost updates and data corruptions that are hard to reproduce and debug.

Two memory accesses in a program are said to *conflict* if they access the same memory location and at least one of them is a write. A program contains a data race if two conflicting accesses can occur concurrently. Figure 1 shows a variation of a data race we found in the Windows kernel. The threads appear to be accessing different fields. However, these bit-fields are mapped to the same word by the compiler and the concurrent accesses result in a data race. In this case, an update to the statistics field possibly hides an update to the status field.

This paper presents *DataCollider,* a tool for dynamically detecting data races in kernel modules. DataCollider is lightweight. It samples a small number of memory accesses for data-race detection and uses code-breakpoint and data-breakpoint[1] facilities available in modern hardware architectures to efficiently perform this sampling. As a result, DataCollider has *no* runtime overhead for non-sampled memory accesses allowing the tool to run with negligible overheads for low sampling rates.

We have implemented DataCollider for the 32-bit Windows kernel running on the x86 architecture, and used it to detect data races in the core kernel and several modules such as the filesystem, the networking stack, the storage drivers, and a network file system. We have found a total of 25 erroneous data races of which 12 have already been fixed at the time of writing. In our experiments, the tool is able to find erroneous data races for sampling rates that incur runtime overheads of less than 5%.

Researchers have proposed multitude of dynamic data-race detectors [1,2,3,4,5,6,7] for user-mode programs. In essence, these tools work by dynamically monitoring the memory accesses and synchronizations performed during a concurrent execution. As data races manifest rarely at runtime, these tools attempt to infer conflicting accesses that *could* have executed concurrently. The tools differ in how they perform this inference, either

[1] Data breakpoints are also called hardware watchpoints.

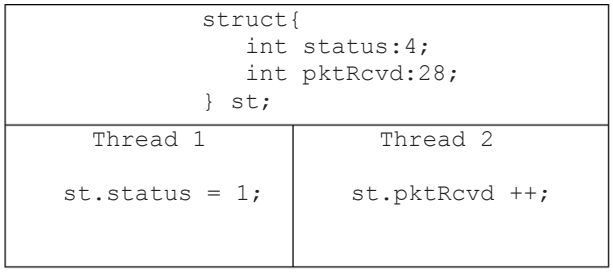


Figure 1: An example of data race. Even though the threads appear to be modifying different variables in the source code, the variables are bit fields mapping to the same integer

using the *happens-before* [8] ordering induced by the synchronization operations [4,5,6] or a *lock-set* based reasoning [1] or a combination of the two [2,3,7]

There are several challenges in engineering a data-race detection tool for the kernel based on previous approaches. First, the kernel-mode code operates at a lower concurrency abstraction than user-mode code, which can rely on clean abstractions of threads and synchronizations provided by the kernel. In the kernel, the same thread context can execute code from a user-mode process, a device interrupt service routine, or a deferred procedure call (DPC). In addition, it is an onerous task to understand the semantics of complex synchronization primitives in order to infer the happens-before relation or lock-sets. For instance, Windows supports more than a dozen locks with different semantics on how the lock holder synchronizes with hardware interrupts, the scheduler, and the DPCs. It is also common for kernel modules to roll-out custom implementations of synchronization primitives.

Second, hardware-facing kernel modules need to synchronize with hardware devices that concurrently modify device state and memory. It is important to design a data-race detection tool that can find these otherwise hard-to-find data races between the hardware and the kernel.

Finally, existing dynamic data-race detectors add prohibitive run-time overheads. It is not uncommon for such tools to incur up to 200x slowdowns [9]. The overhead is primarily due to the need to monitor and process all memory and synchronization operations at run time. Significant engineering effort in building data-race detectors goes in reducing the runtime overhead and the associated memory and log management [9,3]. Replicating these efforts within the constraints of kernel programming is an arduous, if not impossible, task.

```
AtPeriodicIntervals() {
  // determine k based on desired
  // memory access sampling rate
  repeat k times {
    pc = RandomlyChosenMemoryAccess();
    SetCodeBreakpoint( pc );
  }
}

OnCodeBreakpoint( pc ) {
  // disassemble the instruction at pc
  (loc, size, isWrite) = disasm( pc );

  DetectConflicts(loc, size, isWrite);

  // set another code break point
  pc = RandomlyChosenMemoryAccess();
  SetCodeBreakpoint( pc );
}

DetectConflicts( loc, size, isWrite) {
  temp = read( loc, size );

  if ( isWrite )
    SetDataBreakpointRW( loc, size );
  else
    SetDataBreakpointW( loc, size );

  delay();

  ClearDataBreakpoint( loc, size );

  temp' = read( loc, size );

  if( temp != temp' ||
      data breakpoint fired )
    ReportDataRace( );
}
```

Figure 2: The basics of the DataCollider algorithm. Right before a read or write access to shared memory location, chosen at random, DataCollider monitors for any concurrent accesses that conflict with the current access.

Moreover, these tools rely on invasive instrumentation techniques that are difficult to get right on low-level kernel code.

DataCollider uses a different approach to overcome these challenges. The crux of the algorithm is shown in Figure 2. DataCollider samples a small number of memory accesses at runtime by inserting code breakpoints at randomly chosen memory access instructions. When a code breakpoint fires, DataCollider detects data races involving the sampled memory access for a small time window. It simultaneously employs two strategies

to do so. First, DataCollider sets a data breakpoint to trap conflicting accesses by other threads. To detect conflicting writes performed by hardware devices and by processors accessing the memory location through a different virtual address, DataCollider use a *repeated-read* strategy. It reads the value once before and once after the delay. A change in value is an indication of a conflicting write, and hence a data race.

The DataCollider algorithm has two features that make it suitable for kernel data-race detection. First and foremost, it is easy to implement. Barring some implementation details (Section 3), the entire algorithm is shown in Figure 2. In addition, it is entirely oblivious to the synchronization protocols used by the kernel and the hardware, a welcome design point as DataCollider does not have to understand the complex semantics of kernel synchronization primitives.

When the DataCollider finds a data race through the data-breakpoint strategy, it catches both threads "red-handed," as they are about to execute conflicting accesses. This greatly simplifies the debugging of data race reports from DataCollider as the tool can collect useful debugging information, such as the stack trace of the racing threads along with their context information, without incurring this overhead on non-sampled or non-racy accesses.

Not all data races are erroneous. Such *benign* races include races that do not affect the program outcome, such as updates to logging/debugging variables, and races that affect the program outcome in a manner acceptable to the programmer, such as conflicting updates to a low-fidelity counter. DataCollider uses a post-processing phase that prunes and prioritizes the data-race reports before showing them to the user. In our experience with DataCollider, we have observed that only around 10% percentage of data-race reports correspond to real errors, making the post-processing step absolutely crucial for the usability of the tool.

2. Background and Motivation

Shared memory multiprocessors are specifically built to allow concurrent access to shared data. So why do data races represent a problem at all?

The key motivation for data race detection is the empiric fact that programmers most often use synchronization to restrict accesses to shared memory. Data races can thus be an indication of incorrect or insufficient synchronization in the program. In addition, data races can also reveal programming mistakes not directly related to concurrency, such as buffer overruns or use-after-free, which indirectly result in inadvertent sharing of memory.

Another important reason for avoiding data races is to protect the program from the weak memory models of the compiler and the hardware. Both the compiler and hardware can reorder instructions and change the behavior of racy programs in complex and confusing ways [10,11]. Even if a racy program works correctly for the current compiler and hardware configuration, it might fail on future configurations that implement more aggressive memory-model relaxations.

While bugs caused by data races may of course be found using more conventional testing approaches such as stress testing, the latter often fails to provide actionable information to the programmer. Clearly, a data race report including stack traces or data values (or even better, including a core dump that is demonstrating the actual data race) is easier to understand and fix than a silent data corruption that leads to an obscure failure at some later point during program execution.

2.1. Definition of Data Race

There is no "gold standard" for defining data races; several researchers have used the term to mean different things. For our definition, we consulted two respected standards (Posix threads [12] and the drafts of the C++ and C memory model standards [11,10]) and generalized their definitions to account for the particularities of kernel code. Our definition of data race is:

- Two operations that access main memory are called *conflicting* if
  - the physical memory they access is not disjoint,
  - at least one of them is a write, and
  - they are not both synchronization accesses.
- A program *has a data race* if it can be executed on a multiprocessor in such a way that two conflicting memory accesses are performed simultaneously (by processors or any other device).

This definition is a simplification of [11,10] insofar we replaced the tricky notion of "not ordered before" with the unambiguous "performed simultaneously" (which refers to real time).

An important part of our definition is the distinction between synchronization and data accesses. Clearly, some memory accesses participate in perfectly desirable races: for example, a mutex implementation may perform a "release" by storing the value 0 in a shared loca-

tion, while another thread is performing an acquire and reads the same memory location. However, this is not a *data* race because we categorize both of these accesses as synchronization accesses. Synchronization accesses either involve hardware synchronization primitives such as interlocked instructions or use volatile or atomic annotations supported by the compiler.

Note that our definition is general enough to apply to code running in the kernel, which poses some unique problems not found in user-mode code. For example, in some cases data races can be avoided by turning off interrupts; also, processes can exhibit a data race when accessing different virtual addresses that map to the same physical address. We talk more about these topics in Section 2.3.4.

2.2. Precision of Detection

Clearly, we would like data race detection tools to report as many data races as possible without inundating the user with false error reports. We use the following terminology to discuss the precision and completeness of data race detectors. A *missed race* is a data race that the tool does not warn about. A *benign* data race is a data race that does not adversely affect the behavior of the program. Common examples of benign data races include threads racing on updates to logging or statistics variables and threads concurrently updating a shared counter where the occasional incorrect update of the counter does not affect the outcome of the program. On the other hand, a *false* data race is an error report that does not correspond to a data race in the program. Static data-race detection techniques commonly produce false data races due to their inherent inability to precisely reason about program paths, aliased heap objects, and function pointers. Dynamic data-race detectors can report false data races if they do not identify or do not understand the semantics of *all* the synchronizations used by the program.

2.3. Related Work

Researchers have proposed and built a plethora of race detection tools. We now discuss the major approaches and implementation techniques appearing in related work. We describe both happens-before-based and lock-set-based tracking in some detail (Sections 2.3.2 and 2.3.3), before explaining why neither one is very practical for data race detection in the kernel (Section 2.3.4).

2.3.1. Static vs. Dynamic

Data race detection can be broadly categorized into *static* race detection [13,14,15,16,17], which typically analyzes source or byte code without directly executing the program, and *dynamic* race detection [1,2,3,4,5,6,7], which instruments the program and monitors its execution online or offline.

Static race detectors have been successfully applied to large code bases [13,14]. However, as they rely on approximate information, such as pointer aliasing, they are prone to excessive false warnings. Some tools, especially those targeting large code bases, approach this issue by filtering the reported warnings using heuristics [13]. Such heuristics can successfully reduce the false warnings to a tolerable level, but may unfortunately also eliminate correct warnings and lead to missed races. Other tools, targeted towards highly motivated users that wish to interactively prove absence of data races, report all potential races to the user and rely on user-supplied annotations that indicate synchronization disciplines [16,17].

Dynamic data race detectors are less prone to false warnings than static techniques because they monitor an actual execution of the program. However, they may miss races because successful detection might require an error-inducing input and/or an appropriate thread schedule. Also, many dynamic detectors employ several heuristics and approximations that can lead to false alarms.

Dynamic data race detectors can be classified into categories based on whether they model a happens-before relation [6,5,7] (see Section 2.3.2), lock sets [1] (see Section 2.3.3), or both [2,18].

2.3.2. Happens-Before Tracking

Dynamic data race detectors do not just detect data races that actually took place (in the sense that the conflicting accesses were truly simultaneous during the execution), but look for evidence that such a schedule would have been possible for a slightly different timing. Tracking a *happens-before* relation on program events [8] is one way to infer the existence of a racy schedule. This transitive relation is constructed by recording both the ordering of events within a thread and the ordering effects of synchronization operations across threads.

Once we can properly track the happens-before relation, race detection is straightforward: For any two conflicting accesses A and B, we simply check whether A happens-before B, or B happens-before A, or neither. If

neither, we know there exists a schedule where A and B are simultaneous. If properly tracked, happens-before does not lead to any false alarms. However, precise tracking can be difficult to achieve in practice, as discussed in Section 2.3.4.

2.3.3. Lock Sets

When detecting races in programs that follow a strict and consistent locking discipline, using a lock-set approach can provide some benefits. The basic idea is to examine the lock set of each data access (that is, the set of locks held during the access) and then to take for each memory location the intersection of the lock sets of all accesses to it. If that intersection is empty, the variable is not consistently protected by any one lock and a warning is issued.

The main limitation of the lock set approach is that it does not check for true data races but for violations of a specific locking discipline. Unfortunately, many applications (and in particular kernel code) use locking disciplines that are complex and use synchronization other than locks.

Whenever a program departs from a simple locking scheme in any of the above ways, lock-set-based race detectors will be forced to either issue false warnings, or to use heuristics to suppress these warnings. The latter approach is common, especially in the form of state machines that track the "sharing status" of a variable [1,3]. Such heuristics are necessarily imperfect compromises, however (they always fail to suppress some false warnings and always suppress some correct warnings), and it is not clear how to tune them to be useful for a wide range of applications.

2.3.4. Problems with Tracking Synchronizations

Both lock-set and happens-before tracking require a thorough understanding of the synchronization semantics, lest they produce false alarms or miss races. There are two fundamental difficulties we encountered when trying to apply these techniques in the kernel:

- Abstractions that we take for granted in user mode (such as threads) are no longer clearly defined in kernel mode.
- The synchronization vocabulary of kernel code is much richer and may include complicated sequences and ordering mechanisms provided by the hardware.

For example, interrupts and interrupt handlers break the thread abstraction, as the handler code may execute in a thread context without being part of that thread in a logical sense. Similar problems arise when a thread calls into the kernel scheduler. The code executing in the scheduler is not logically part of that same thread.

Another example illustrating the difficulty of modeling synchronization inside the kernel are DMA accesses. Such accesses are not executing inside a thread (in fact, they are not even executing on a processor). Clearly, traditional monitoring techniques have a problem because they cannot "instrument" the DMA access.

Similar case holds for interrupt processing. For example, code may first write some data and then raise an interrupt, and then the same data is read by an interrupt handler. Lock sets would report a false alarm because the data is not locked. But even happens-before techniques are problematic, because they would need to precisely track the causality between the instruction that set the interrupt and the interrupt handler.

For these reasons, we decided to employ a design that entirely avoids modeling the happens-before ordering or lock-sets. As our results show, somewhat surprisingly, neither one is required to build an effective data race detector.

2.3.5. Sampling to Reduce Overhead

To detect races, dynamic data race detectors need to monitor the synchronizations and memory accesses performed at runtime. This is typically done by instrumenting the code and inserting extra monitoring code for each data access. As the monitoring code executes at every memory access, the overhead can be quite substantial.

One way to ameliorate this issue is to exclude some data accesses from processing. Prior work has identified several promising strategies: adaptive sampling that backs off hot locations [5] (the idea is that for such locations the monitoring can be less frequent and still detect races), or perform the full monitoring only for a fixed fraction of the time [4] (the idea is that the probability of catching a race is roughly proportional to this fraction multiplied by the number of times the race repeats). But these techniques still suffer from the cost of sampling, performed at every memory access. DataCollider avoids this problem by using hardware breakpoint mechanisms.

3. DataCollider Implementation

This section describes the implementation of the DataCollider algorithm for the Windows kernel on the x86 architecture. The implementation heavily uses the code and data breakpoint mechanisms available on x86. The techniques described in this paper can be extended to other architectures and to user-mode code. But we have not pursued this direction in this paper.

Figure 2 describes the basics of the DataCollider algorithm. DataCollider uses the sampling algorithm, described in Section 3.1, to process a small percentage of memory accesses for data-race detection. For each of the sampled memory accesses, DataCollider uses a conflict detection mechanism, described in Section 3.2, to find data races involving the sampled access. After detecting data races, DataCollider uses several heuristics, described in Section 3.3, to prune benign data races.

3.1. The Sampling Algorithm

There are several challenges in designing a good sampling algorithm for data-race detection. First, data races involve two memory accesses both of which need to be sampled to detect the race. If memory accesses are sampled independently, then the probability of finding the data race is a product of the individual sampling probabilities. DataCollider avoids this multiplicative effect by sampling the first access and using a data breakpoint to trap the second access. This allows DataCollider to be effective at low sampling rates.

Second, data races are rare events – most executed instructions do not result in a data race. The sampling algorithm should weed out the small percentage of racing accesses from the majority of non-racing accesses. The key intuition behind the sampling algorithm is that if a program location is buggy and fails to use the right synchronization when accessing shared data, then every dynamic execution of that buggy code is likely to participate in a data race. Accordingly, DataCollider performs *static* sampling of program locations rather than *dynamic* sampling of executed instructions. A static sampler provides equal preference to rarely execution instructions (which are likely to have bugs hidden in them) and frequently executed instructions.

3.1.1. Static Sampling Using Code Breakpoints

The static sampling algorithm works as follows. Given a program binary, DataCollider disassembles the binary to generate a *sampling set* consisting of all program locations that access memory. The tool currently requires the debugging symbols of the program binary to perform this disassembly. This requirement can be relaxed by using sophisticated disassemblers [19] in the future.

DataCollider performs a simple static analysis to identify instructions that are guaranteed to only touch thread-local stack locations and removes them from the sampling set. Similarly, DataCollider removes synchronizing instructions from the sampling set by removing instructions that accesses memory locations tagged as "volatile" or those that use hardware synchronization primitives, such as interlocked. This prevents DataCollider from reporting races on synchronization variables. However, DataCollider can still detect a data race between a synchronization access and a regular data access, if the latter is in the sampling set.

DataCollider samples program locations from the sampling set by inserting code breakpoints. The initial breakpoints are set at a small number of program locations chosen uniformly randomly from the sampling set. If and when a code breakpoint fires, DataCollider performs conflict detection for the memory access at that breakpoint. Then, DataCollider choses another program location uniformly randomly from the sampling set and sets a breakpoint at that location.

This algorithm uniformly samples all program locations in the sampling set irrespective of the frequency with which the program executes these locations. This is because the choice of inserting a code breakpoint is performed uniformly at random for all locations in the sampling set. Over a period of time, the breakpoints will tend to reside at rarely executed program locations, increasing the likelihood that those locations are sampled the next time they execute.

If DataCollider has information on which program locations are likely to participate in a race, either through user annotations or through prior analysis [20] then the tool can prioritize those locations by biasing their selection from the sampling set.

3.1.2. Controlling the Sampling Rate

While the program cannot affect the sampling distribution over program locations, the sampling *rate* is intimately tied to how frequently the program executes locations with a code breakpoint. In the worst case, if all of the breakpoints are set on dead code, DataCollider will stop performing data-race detection altogether. To avoid this and to better control the sampling rate, DataCollider periodically checks the number of breakpoints fired every second, and adjusts the number of

breakpoints set in the program based on whether the experienced sampling rate is higher or lower than the target rate.

3.2. Conflict-Detection

As described in the previous section, DataCollider picks a small percentage of memory accesses as likely candidates for data-race detection. For these sampled accesses, DataCollider pauses the current thread waiting to see if another thread makes a conflicting access to the same memory location. It uses two strategies: data breakpoints and repeated-reads. DataCollider uses these two strategies simultaneously as each complements the weaknesses of the other.

3.2.1. Detecting Conflicts with Data Breakpoints

Modern hardware architectures provide a facility to trap when a processor reads or writes a particular memory location. This is crucial for efficient support for data breakpoints in debuggers. The x86 hardware supports four data breakpoint registers. DataCollider uses them to effectively monitor possible conflicting accesses to the currently sampled access.

When the current access is a write, DataCollider instructs the processor to trap on a read or write to the memory location. If the current access is a read, DataCollider instructs the processor to trap only on a write, as concurrent reads to the same location do not conflict. If no conflicting accesses are detected, DataCollider resumes the execution of the current thread after clearing the data breakpoint registers.

Each processor has a separate data breakpoint register. DataCollider uses an inter-processor interrupt to update the break points on all processors atomically. This also synchronizes multiple threads attempting to sample different memory locations concurrently.

An x86 instruction can access variable sized memory. For 8, 16, or 32-bit accesses, DataCollider sets a breakpoint of the appropriate size. The x86 processor traps if another instruction accesses a memory location that overlaps with a given breakpoint. Luckily, this is precisely the semantics required for data-race detection. For accesses that span more than 32 bits, DataCollider uses more than one breakpoint up to the maximum available of four. If DataCollider runs out of breakpoint registers, it simply resorts to the repeated-read strategy discussed below.

When a data breakpoint fires, DataCollider has successfully detected a race. More importantly, it has caught the racing threads "red handed" – the two threads are at the point of executing conflicting accesses to the same memory location.

One particular shortcoming of data breakpoint support in x86 that we had to work around was the fact that, when paging is enabled, x86 performs the breakpoint comparisons based on the virtual address and has no mechanism to modify this behavior. Two concurrent accesses to the same virtual addresses but different physical addresses do not race. In Windows, most of the kernel resides in the same address space with two exceptions.

Kernel threads accessing the user address space cannot conflict if the threads are executing in the context of different processes. If a sampled access lies in the user address space, DataCollider does not use breakpoints and defaults to the repeated-read strategy.

Similarly, a range of kernel-address space, called *session memory*, is mapped to different address spaces based on the session the process belongs to. When a sampled access lies in the session memory space, DataCollider sets a data breakpoint but checks if the conflicting accesses belong to the same session before reporting the conflict to the user.

Finally, a data breakpoint will miss conflicts if a processor uses a different virtual address mapped to the same physical address as the sampled access. Similarly, data breakpoints cannot detect conflicts arising from hardware devices directly accessing memory. The repeated-read strategy discussed below covers all these cases.

3.2.2. Detecting Conflicts with Repeated Reads

The repeated-read strategy relies on a simple insight: if a conflicting write changes the value of a memory location, DataCollider can detect this by repeatedly reading the memory location checking for value changes. An obvious disadvantage of this approach is that it cannot detect conflicting reads. Similarly, it cannot detect multiple conflicting writes the last of which writes the same value as the initial value. Despite these shortcomings, we have found this strategy to be very useful in practice. This is the first strategy we implemented (as it is easier to implement than using data breakpoints) and we were able to find several kernel bugs with this approach.

However, repeated-reads strategy catches only one of the two threads "red-handed." This makes it harder to debug data races, as one does not know which thread or device was responsible for the conflicting write. This was our prime motivation for using data breakpoints.

3.2.3. Inserting Delays

For a sampled memory access, DataCollider attempts to detect a conflicting access to the same memory location by delaying the thread for a short amount of time. For DataCollider to be successful, this delay has to be long enough for the conflicting access to occur. On the other hand, delaying the thread for too long can be dangerous especially if the thread holds some resource crucial for the proper functioning of the entire system. In general, it is impossible to predict how long to insert the delay. After experimenting with many values, we chose the following delay algorithm.

Depending on the IRQL (Interrupt Request Level) of the executing thread, DataCollider delays the thread for a preset maximum amount of time. At IRQLs higher than the DISPATCH level (the level at which the kernel scheduler operates), DataCollider does not insert any delay. We considered inserting a small window of delay at this level to identify possible data races between interrupt service routines. But we did not expect that DataCollider would be effective at short delays.

Threads running at the DISPATCH level cannot yield the processor to another thread. As such, the delay is simply a busy loop. We currently delay threads at this level for a random amount of time less than 1 ms. For lower IRQLs, DataCollider delays the thread for a maximum of 15 ms by spinning in a loop that yields the current time quantum. During this loop, the thread repeatedly checks to see if other threads are making progress by inspecting the rate at which breakpoints fire. If progress is not detected, the waiting thread prematurely stops its wait.

3.3. Dealing with Benign Data Races

Research on data-race detection has amply noted the fact that not all data races are erroneous. A practical data-race detection tool should effectively prune or deprioritize these benign data races when reporting to the user. However, inferring whether or not a data race is benign can be tricky and might require deep understanding of the program. For instance, a data race between two concurrent non-atomic counter updates might be benign if the counter is a statistic variable whose fidelity is not important to the behavior of the program. However, if the counter is used to maintain the number of references to a shared object, then the data race could lead to a memory leak or a premature free of the object.

During the initial runs of the tool, we found that around 90% of the data-race reports are benign. Inspecting these we identified the following patterns that can be identified through simple static and/or dynamic analysis and incorporated them in a post-process pruning phase.

Statistics Counters: Around half of the benign data races involved conflicting updates to counters that maintain various statistics about the program behavior [21]. These counters are not necessarily write-only and could affect the control flow of the program. A common scenario is to use these counter value to perform periodic computation such as flushing a log buffer. If DataCollider reports several data races involving an increment instruction and the value of the memory location consistently increases across these reports, then the pruning phase tags these data races as statistics-counter races. Checking for an increase in memory values helps the pruning phase in distinguishing these statistics counters from reference counters that are usually both incremented and decremented.

Safe Flag Updates: The next prominent class of benign races involves a thread reading a flag bit in a memory location while another thread updates a different bit in the same memory location. By analyzing few memory instructions before and after the memory access, the pruning phase identifies read-write conflicts that involve different bits. On the other hand, write-write conflicts can result in lost updates (as shown in Figure 1) and are not tagged as benign.

Special Variables: Some of the data races reported by DataCollider involve special variables in the kernel where races are expected. For instance, Windows maintains the current time in a variable, which is read by many threads while being updated by the timer interrupt. The pruning phase has a database of such variables and prunes races involving these variables.

While it is possible to design other patterns that identify benign data races, one has to tradeoff the benefit of the pruning achieved with the risk of missing real data races. For instance, we initially designed a pattern to classify two writes that write the same value as benign. However, very few data-race reports matched this property. On the other hand, Figure 4 shows an example of a harmful data-race that we found involving two such writes.

Also, we have made an explicit decision to make the benign data races available to the user, but deprioritized

Data Races Reported	Count
Fixed	12
Confirmed and Being Fixed	13
Under Investigation	8
Harmless	5
Total	38

Figure 3: Bugs reported to the developers after excluding benign data-race reports.

against races that are less likely to be benign. Some of our users are interested in browsing through the pruned benign races to identify potential portability problems and memory-model issues in their code. We also found an instance where a benign race, despite being harmless, indicated unintended sharing in the code and resulted in a design change.

4. Evaluation

There are two metrics for measuring the success of a data-race detection tool. First, is it able to find data races that programmers deem important enough to fix? Second, is it able to scale to a large system, which in our case is the Windows operating system, with reasonable runtime overheads? This section presents a case for an affirmative claim on these two metrics.

4.1. Experimental Setup

For the discussion in this section, we applied DataCollider on several modules in the Windows operating system. DataCollider has been has been used on class drivers, various PnP drivers, local and remote file system drivers, storage drivers, and the core kernel executive itself. We are successfully able to boot the operating system with DataCollider and run existing kernel stress tests.

4.2. Bugs Found

Figure 3 presents the data race reports produced by the different versions of DataCollider during its entire development. We reported a total 38 data-race reports to the developers. This figure does not reflect the number of benign data races pruned heuristically and manually. We defer the discussion of benign data races to Section 4.4.

Of these 38 reports, 25 have been confirmed as bugs and 12 of which have already been fixed. The developers indicated that 5 of these are indeed harmless. For instance, one of the benign data races results in a driver issuing an idempotent request to the device. While this could result in a performance loss, the expected frequency of the data race did not justify the cost of adding synchronization in the common case. Identifying such benign races requires intimate knowledge of the code and would not be possible without the programmers help.

As DataCollider naturally delays the racing access that temporally occurs first, it is likely to explore both outcomes of the race. Despite this, only one of the 38 data races crashed the kernel in our experiments. This indicates that the effects of an erroneous data race are not immediately apparent for the particular input or the hardware configuration of the current run.

We discuss two interesting error reports below

4.2.1. A Boot Hang Caused by a Data Race

A hardware vendor was consistently seeing a kernel hang at boot-up time. This was not reproducible in any of the in-house machine configurations, till the vendor actually shipped the hardware to the developers. After inspecting the hang, a developer noticed a memory corruption in a driver that could be a result of a race condition. When analyzing the driver in question, DataCollider found the data race in an hour of testing on a regular in-house machine (in which the kernel did not hang). Once the source of the corruption was found (performing a status update non-atomically), the bug was immediately fixed.

```
void AddToCache() {
   // ...
   A: x &= ~(FLAG_NOT_DELETED);
   B: x |= FLAG_CACHED;

   MemoryBarrier();
   // ...
}

AddToCache();
assert( x & FLAG_CACHED );
```

Figure 4: An erroneous data race when the `AddToCache` function is called concurrently. Though the data race appears benign, as the conflicting accesses "write the same values," the `assert` can fail on some thread schedules.

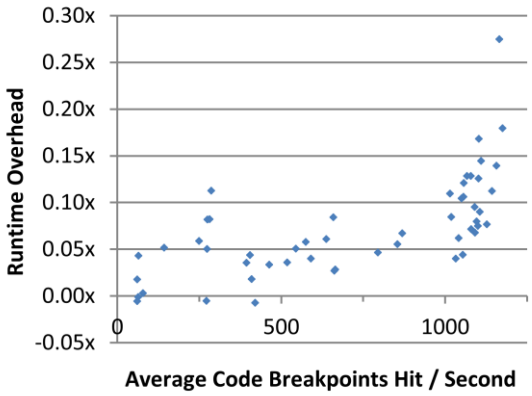


Figure 5: Runtime overhead of DataCollider with increasing sampling rate, measured in terms of the number of code breakpoints firing per second. The overhead tends to zero as the sampling rate is reduced, indicating that the tool has negligible base overhead.

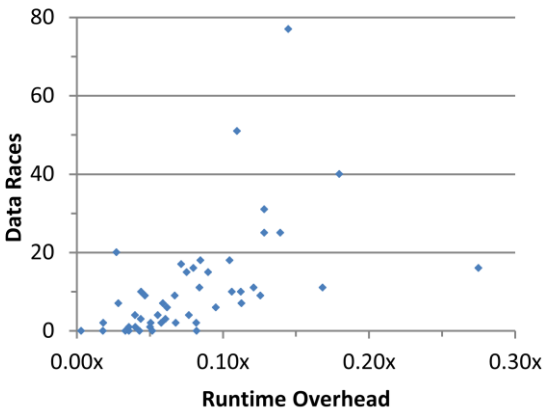


Figure 6: The number of data races, uniquely identified by the pair of racing program locations, with the runtime overhead. DataCollider is able to report data race even under overheads under 5%

4.2.2. A Not-So-Benign Data Race

Figure 4 shows an erroneous data race. The function `AddToCache` performs two non-atomic updates to the flag variable. DataCollider produced an error report with two threads simultaneously updating the flag at location B. Usually, two instructions writing the same values is a good hint that the data race is benign. However, the presence of the memory barrier indicated that this report required further attention – the developer was well aware of consequences of concurrency and the rest of the code relied on crucial invariants on the flag updates. When we reported this data race to the developer he initially tagged it as benign. On further discussion, we discovered that the code relied on the invariant that the `CACHED` bit is set after a call to `AddToCache`. The data race can break this invariant when a concurrent thread overwrites `CACHED` bit when performing the update at A, but gets preempted before setting the bit at B.

4.2.3. How Fixed

While data races can be hard to find and result in mysterious crashes, our experience is that most are relatively easy to fix. Of the 12 bugs, 3 were the result of missing locks. The developer could easily identify the locking discipline that was meant to be followed, and could decide which lock to add without the fear of a deadlock. 6 data races were the fixed by using an atomic instructions, such as interlocked increment, to make a read-modify-write to a shared variable. 2 bugs were a result of unintended sharing and were fixed by making the particular variable thread local. Finally, one bug indicated a broken design due to a recent refactoring and resulted in a design change.

4.3. Runtime Overhead

Users have an inherent aversion to dynamic analysis tools that add prohibitive runtime overheads. The obvious reason is the associated wastage of test resources – a slowdown of ten means that only one-tenth the amount of testing can be done with a given amount of resources. More importantly, runtime overheads introduced by a tool can affect the real-time execution of the

Data Race Category		Count
Benign – Heuristically Pruned	Statistic Counter	52
	Safe Flag Update	29
	Special Variable	5
	Subtotal	86
Benign – Manually Pruned	Double-check locking	8
	Volatile	8
	Write Same Value	1
	Other	1
	Subtotal	18
Real	Confirmed	5
	Investigating	4
	Subtotal	9
Total		113

Figure 7: Categorization of data races found by DataCollider during kernel stress.

program. The operating system could start a recovery action if a device interrupt takes too long to finish. Or a test harness can incorrectly tag a kernel-build faulty if it takes too long to boot.

To measure the runtime overhead of DataCollider, we repeatedly measured the time taken for the boot-shutdown sequence for different sampling rates and compared against a baseline Windows kernel running without DataCollider. These experiments where done on the x86 version of Windows 7 running on a virtual machine with 2 processors and 512 MB memory. The host machine is an Intel Core2-Quad 2.4 GHz machine with 4 GB memory running Windows Server 2008. The guest machine was limited to 50% of the processing resources of the host. This was done to prevent any background activity on the host from perturbing the performance of the guest.

Figure 5 shows the runtime overhead of DataCollider for different sampling rates, measured by the average number of code breakpoints fired per second during the run. As expected, the overhead increases roughly linearly with the sampling rate. More interestingly, as the sampling rate tends to zero, DataCollider's overhead reaches zero. This indicates that DataCollider can be "always on" in various testing and deployment scenarios, allowing the user to tune the overhead to any acceptable limit.

Figure 6 shows the number of data races detected for different runtime costs. DataCollider is able to detect data races even for overheads less than 5% indicating the utility of the tool at low overheads.

4.4. Benign Data Races

Finally, we performed an experiment to measure the efficacy of our pruning algorithm for benign data races. The results are shown in Figure 7. We enabled DataCollider while running kernel stress tests for 2 hours sampling at approximately 1000 code breakpoints per second. DataCollider found a total of 113 unique data races. The patterns described in Section 3.3 can identify 86 (76%) of these as benign errors. We manually (and painfully) triaged these reports to ensure that these races were truly benign. Of the remaining races, we manually identified 18 as not erroneous. 8 of them involved the double-checked locking idiom, where a thread performs a racy read of a flag without holding a lock, but reconfirms the value after acquiring the lock. 8 were accesses to volatile variables that DataCollider's analysis was unable to infer the type of. These reports can be avoided with a more sophisticated analysis for determining the program types. This table demonstrates that a significant percentage of benign data races can be heuristically pruned without risks of missing real data races. During this process, we found 9 potentially harmful data races of which 5 have already been confirmed as bugs.

5. Conclusion

This paper describes DataCollider, a lightweight and effective data-race detector specifically designed for low-level systems code. Using our implementation of DataCollider for the Windows operating system, we have found to date 25 erroneous data races of which 12 are already fixed.

We would like to thank our shepherd Junfeng Yang and all our anonymous reviewers for valuable feedback on the paper.

References

[1] Stefan Savage, Michael Burrows, Greg Nelson, and Patrick Sobalvarro, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 391-411, 1997.

[2] Robert O'Callahan and Jong-Deok Choi, "Hybrid Dynamic Data Race Detection," *SIGPLAN Not.*, vol. 38, no. 10, pp. 167-178, 2003.

[3] Yuan Yu, Tom Rodeheffer, and Wei Chen, "RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking," in *Symposium on Operating System Principles (SOSP)*, 2005, pp. 221-234.

[4] Michael D Bond, Katherine E Coons, and Kathryn S McKinley, "PACER: Proportional Detection of Data Races," in *Programming Languages Design and Implementation (PLDI)*, 2010.

[5] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasami, "LiteRace: Effective Sampling for Lightweight Data-Race Detection," in *Programming Language Design and Implementation*, 2009, pp. 134-143.

[6] Cormac Flanagan and Stephen N Freund, "FastTrack: Efficient and Precise Dynamic Race Detection," in *Programming Language Design and Implementation*, 2009, pp. 121-133.

[7] E Pozniansky and A Schuster, "MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs," *Concurrency and Computation: ractice and Experience*, vol. 19, no. 3, pp. 327-340, 2007.

[8] Leslie Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558-565, 1978.

[9] Paul Sack, Brian E Bliss, Zhiqiang Ma, Paul Petersen, and Josep Torrellas, "Accurate and Efficient Filtering for the Intel Thread Checker Race Detector," in *Workshop on Architectural and System Support for Improving Software Dependability*, 2006, pp. 34-41.

[10] Hans Boehm and Sarita Adve, "Foundations of the C++ Concurrency Memory Model," HP Labs, Technical Report HPL-2008-56 , 2008.

[11] Hans Boehm. (2009, Sep.) N1411: Memory Model Rationale. [Online]. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1411.htm

[12] IEEE, POSIX.1c, Threads extensions, 1995, IEEE Std 1003.1c.

[13] Dawson Engler and Ken Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," in *Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 237-252.

[14] Mayur Naik, Alex Aiken, and John Whaley, "Effective Static Race Detection for Java," in *Programming Language Design and Implementation (PLDI)*, 2006, pp. 308-319.

[15] Cormac Flanagan and Stephen Freund, "Type-Based Race Detection for Java," in *Programming Language Design and Implementation (PLDI)*, Vancouver, 2000, pp. 219-232.

[16] Zachary Anderson, David Gay, and Mayur Naik, "Lightweight Annotations for Controlling Sharing in Concurrent Data Structures," in *Programming Language Design and Implementation (PLDI)*, Dublin, 2009.

[17] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard, "Ownership Types for Safe Programming: Preventing Data Races and Deadlocks," in *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2002, pp. 211-230.

[18] A Dinning and E Schonberg, "Detecting access anomalies in programs with critical sections," in *Workshop on Parallel and Distributed Debugging*, 1991, pp. 85-96.

[19] The IDA Pro Disassembler and Debugger. [Online]. http://www.hex-rays.com/idapro/

[20] Koushik Sen, "Race Directed Random Testing of Concurrent Programs," in *Programming Language Design and Implementation (PLDI'08)*, 2008, pp. 11-21.

[21] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder, "Automatically Classifying Benign and Harmful Data Races Using Replay Analysis," in *Programming Language Design and Implementation (PLDI '07)*, 2007, pp. 22-31.

[22] Donald E. Knuth, *The Art of Computer Programming, Volume 2.*: Addison-Wesley Longman, 1997.

[23] Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder P. Singh, and Anoop Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *ISCA '95: International Symposium on Computer architecture*, 1995, pp. 24-26.

[24] Amitabh Srivastava and Alan Eustace, "ATOM: A System for Building Customized Program Analysis Tools," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994, pp. 196-205.

Ad Hoc Synchronization Considered Harmful

*Weiwei Xiong†, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, Zhiqiang Ma**

*University of California, San Diego †University of Illinois at Urbana-Champaign *Intel*

Abstract

Many synchronizations in existing multi-threaded programs are implemented in an ad hoc way. The first part of this paper does a comprehensive characteristic study of ad hoc synchronizations in concurrent programs. By studying 229 ad hoc synchronizations in 12 programs of various types (server, desktop and scientific), including Apache, MySQL, Mozilla, etc., we find several interesting and perhaps *alarming* characteristics: (1) Every studied application uses ad hoc synchronizations. Specifically, there are 6–83 ad hoc synchronizations in each program. (2) Ad hoc synchronizations are error-prone. *Significant percentages (22–67%) of these ad hoc synchronizations introduced bugs or severe performance issues*. (3) Ad hoc synchronization implementations are diverse and many of them cannot be easily recognized as synchronizations, i.e. have poor readability and maintainability.

The second part of our work builds a tool called **SyncFinder** to automatically identify and annotate ad hoc synchronizations in concurrent programs written in C/C++ to assist programmers in porting their code to better structured implementations, while also enabling other tools to recognize them as synchronizations. Our evaluation using 25 concurrent programs shows that, on average, SyncFinder can automatically identify 96% of ad hoc synchronizations with 6% false positives.

We also build two use cases to leverage SyncFinder's auto-annotation. The first one uses annotation to detect 5 deadlocks (including 2 new ones) and 16 potential issues missed by previous analysis tools in Apache, MySQL and Mozilla. The second use case reduces Valgrind data race checker's false positive rates by 43–86%.

1 Introduction

Synchronization plays an important role in concurrent programs. Recently, partially due to realization of multi-core processors, much work has been conducted on synchronization in concurrent programs. For example, various hardware/software designs and implementations have been proposed for transactional memory (TM) [37, 13, 30, 40] as ways to replace the cumbersome "lock" operations. Similar to TM, some new language constructs [46, 7, 12] such as Atomizer [12] have also been proposed to address the atomicity problem. On a different but related note, various tools such as AVIO [27], CHESS [31], CTrigger [36], ConTest [6] have been built to detect or expose atomicity violations and data races in concurrent programs. In addition to atomicity synchronization, condition variables and monitor mechanisms have also been studied and used to ensure certain execution order among multiple threads [14, 16, 22].

So far, most of the existing work has targeted only the synchronizations implemented in a modularized way, i.e., directly calling some primitives such as "lock/unlock" and "cond_wait/cond_signal" from standard POSIX thread libraries or using customized interfaces implemented by programmers themselves. Such synchronization methods are easy to recognize by programmers, or bug detection and performance profiling tools.

Unfortunately, besides modularized synchronizations, programmers also use their own ad hoc ways to do synchronizations. It is usually hard to tell ad hoc synchroniztions apart from ordinary thread-local computations, making it difficult to recognize by other programmers for maintenance, or tools for bug detection and performance profiling. We refer to such synchronization as *ad hoc synchronization*. If a program defines its own synchronization primitives as functional calls and then uses these functions throughout the program for synchronization, then we do not consider these primitives as ad hoc, since they are well modularized.

Ad hoc synchronization is often used to ensure an intended execution order of certain operations. Specifically, instead of calling"cond_wait()" and "cond_signal()" or other synchronization primitives, programmers often use *ad hoc loops* to synchronize with some shared variables, referred to as *sync variables*. According to programmers' comments, they are implemented this way due to either flexibility or performance reasons.

Figure 1(a)(b)(c)(d) show four real world examples of ad hoc synchronizations from MySQL, Mozilla, and OpenLDAP. In each example, a thread is waiting for some other threads by repetitively checking on one or more shared variables, i.e. sync variables. Each case has its own specific implementation, and it is also not obviously apparent that a thread is synchronizing with another thread.

Unfortunately, there have been few studies on ad hoc

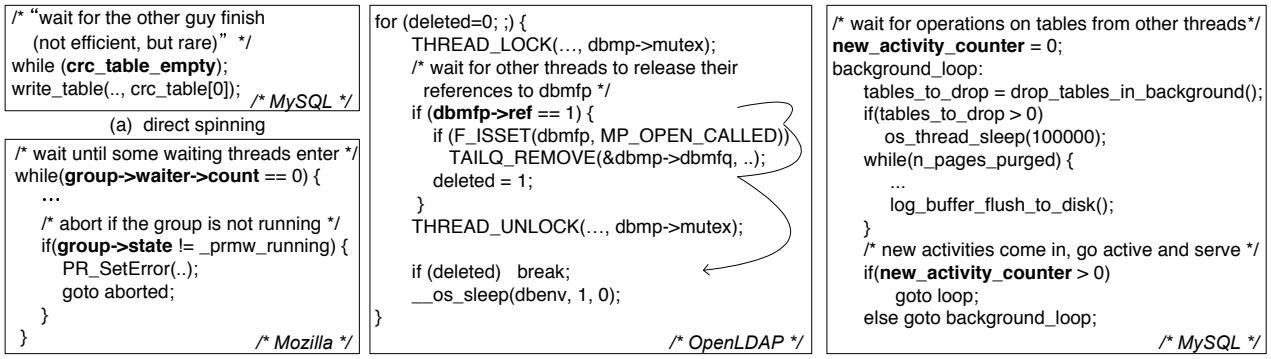


Figure 1: **Real world examples of ad hoc synchronizations**. Sync variables are highlighted using bold fonts. Example (a) directly spins on the sync variable; (b) checks more than one sync variables, (c) takes a certain control path to exit after checking a sync variable, (d) performs some useful work inside the waiting loop.

synchronization. It is unclear how commonly it is used, how programmers implement it, what issues are associated with it, whether it is error-prone or not.

1.1 Contribution 1: Ad Hoc Synchronization Study

In the first part of our work, we conduct a "forensic investigation" of 229 ad hoc synchronizations in 12 concurrent programs of various types (server, desktop and scientific), including Apache, MySQL, Mozilla, OpenLDAP, etc. The goal of our study is to understand the characteristics and implications of ad hoc synchronization in existing concurrent programs.

Our study has revealed several interesting, *alarming* and quantitative characteristics as follows:

(1) Every studied concurrent program uses ad hoc synchronization. More specifically, there are 6–83 ad hoc synchronizations implemented using ad hoc loops in each of the 12 studied programs. The fact that programmers often use ad hoc synchronization is likely due to two primary reasons: (i) Unlike typical atomicity synchronization, when coordinating execution order among threads, the intended synchronization scenario may vary from one to another, making it hard to use a common interface to fit every need (more discussion follows below and in Section 2); (ii) Performance concerns make some of the heavy-weight synchronization primitives less applicable.

(2) Although almost all ad hoc synchronizations are implemented using loops, the implementations are diverse, making it hard to manually identify them among the thousands of computation loops. For example, Figure 1(a) directly spins on a shared variable; Figure 1(b) has multiple exit conditions; Figure 1(c) shows the exit condition indirectly depends on the sync variable and needs complicated calculation to determine whether to exit the loop; Figure 1(d) synchronizes on program states and performs useful work while checking whether the remote thread has changed the states or not. Such characteristic may partially explain why programmers use ad hoc synchronizations. More discussion and examples are in Section 2.

Apps.	#ad hoc sync	#buggy sync
Apache	33	7 (22%)
OpenLDAP	15	10 (67%)
Cherokee	6	3 (50%)
Mozilla-js	17	5 (30%)
Transmission	13	8 (62%)

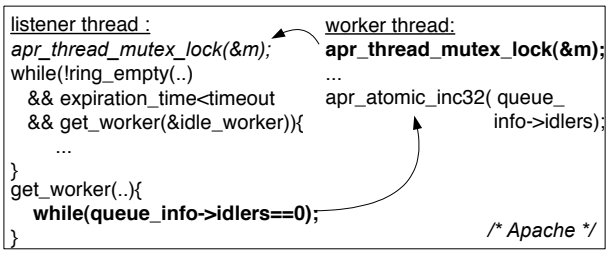
Table 1: **Percentages of ad hoc synchronizations that had introduced bugs according to the bugzilla databases and changelogs of the applications.**

(3) Ad hoc synchronizations are error-prone. Table 1 shows that among the five software systems we studied, signficant percentages (22-67%) of ad hoc synchronizations introduced bugs. Although some experts may expect such results, our study is among the first to provide some quantitative results to back up this observation.

Ad hoc synchronization can easily introduce deadlocks or hangs. As shown on Figure 2, Apache had a deadlock in one of its ad hoc synchronizations. It holds a mutex while waiting on a sync variable "queue_info→idlers". Figure 3 shows another deadlock example in MySQL, which has never been reported previously. More details and the real world examples are in Section 2.

Because they are different from deadlocks caused by locks or other synchronization primitives, deadlocks involving ad hoc synchronizations are very hard to detect using existing tools or model checkers [11, 43, 24]. These tools cannot recognize ad hoc synchronizations unless these synchronizations are annotated manually by programmers or automatically by our SyncFinder described in section 1.2. For the same reason, it is also hard for concurrency testing tools such as ConTest [6] to expose these deadlock bugs during testing.

Furthermore, ad hoc synchronizations also have problems interacting with modern hardware's weak memory



change log: "Never hold mutex while calling blocking operations"

Figure 2: **A deadlock introduced by an ad hoc synchronization in Apache.**

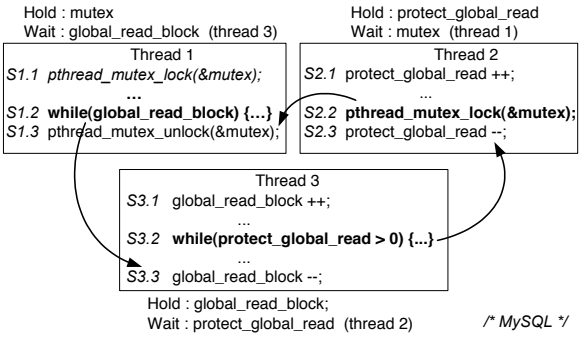


Figure 3: **A deadlock caused by a circular wait among three threads (This is a new deadlock detected by our deadlock detector leveraging SyncFinder's auto-annotation)**. Thread 2 is waiting at S2.2 for the lock to be released by thread 1; thread 1 is waiting at S1.2 for thread 3 to decrease the counter at S3.3; and thread 3 is waiting at S3.2 for thread 2 to decrease another counter at S2.3.

consistency model and also with some compiler optimizations, e.g. loop invariant hoisting (discussed further in Section 2).

By studying the comments associated with ad hoc synchronizations, we found that some programmers knew their implementations might not be safe or optimal, but they still decided to keep their ad hoc implementations.

(4) Ad hoc synchronizations significantly impact the effectiveness and accuracy of various bug detection and performance tuning tools. Since most bug detection tools cannot recognize ad hoc synchronizations, they can miss many bugs related to those synchronizations, as well as introduce many false positives (details and examples in Section 2). For the same reason, performance profiling and tuning tools may confuse ad hoc synchronizations for computation loops, thus generating inaccurate or even misleading results.

1.2 Contribution 2: Identifying Ad Hoc Synchronizations

Our characteristic study on ad hoc synchronization reveals that ad hoc synchronization is often harmful with respect to software correctness and performance. The first step to address the issues raised by ad hoc synchronization is to identify and annotate them, similar to the way that type annotation helps Deputy [9] and SafeDrive [50] to identify memory issues in Linux. Specifically, if ad hoc synchronizations are annotated in concurrent programs, (1) static or dynamic concurrency bug (e.g. data race and deadlock) detectors can leverage such annotations to detect more bugs and prune more false positives caused by ad hoc synchronizations; (2) performance tools can be extended to capture bottlenecks related to these synchronizations; (3) new programming language/model designers can study ad hoc synchronizations to design or revise language constructs; (4) programmers can port such ad hoc synchronizations to more structured implementations.

Unfortunately, ad hoc synchronizations are very hard and time-consuming to recognize and annotate manually. Partly because of this, although some annotation languages for synchronizations like Sun Microsystems' Lock_Lint [2] have been available for several years, they are rarely used, even in Sun's own code [35]. Furthermore, manual examination is also error-prone. Figure 4 shows a MySQL ad hoc synchronization example that we missed during the manual identification we conducted for our characteristic study. Fortunately, our automatic identification tool SyncFinder found it. We overlooked this example because of the complicated nested "goto" loops.

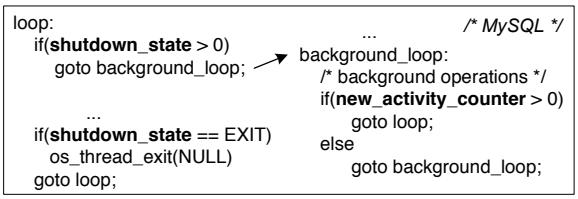


Figure 4: **An ad hoc synchronization missed in our manual identification process of our characteristic study but is identified by our auto-identification tool, SyncFinder**. The interlocked "goto" loops can easily be missed by manual identification (Figure 1(d) shows more detailed code).

Motivated by the above reasons, the second part of our work involved building a tool called **SyncFinder** to automatically identify and annotate ad hoc synchronizations in concurrent programs. SyncFinder statically analyzes source code using inter-procedural, control and data flow analysis, and leverages several of our observations and insights gained from our study to distinguish ad hoc synchronizations apart from thousands of computation loops.

We evaluate SyncFinder with 25 concurrent programs including the 12 used in our characteristic study and 13 others. SyncFinder automatically identifies and annotates 96% of ad hoc synchronization loops with 6% false positives on average.

To demonstrate the benefits of auto-annotation of ad hoc synchronizations by SyncFinder, we design and evaluate two use cases. In the first use case, we build a simple wait-inside-critical-section detector, which can iden-

Apps.	Desc.	LOC.	Total loops	**Ad hoc loops**
Apache 2.2.14	Web server	228K	1462	33
MySQL 5.0.86	Database server	1.0M	4265	83
OpenLDAP 2.4.21	LDAP server	272K	2044	15
Cherokee 0.99.44	Web server	60K	748	6
Mozilla-js 0.9.1	JS engine	214K	848	17
PBZip2 2-1.1.1	Parallel bzip2	3.6K	45	7
Transmission 1.83	BitTorrent client	96K	1114	13
Radiosity	SPLASH-2	14K	80	12
Barnes	SPLASH-2	2.3K	88	7
Water	SPLASH-2	1.5K	84	9
Ocean	SPLASH-2	4.0K	339	20
FFT	SPLASH-2	1.0K	57	7

Table 2: **The number of ad hoc synchronizations in concurrent programs we studied.** Ad hoc sync is implemented with an ad hoc loop using shared variables (i.e., sync variables) in it.

tify deadlock and bad programming practices involving ad hoc synchronizations. In our evaluation, our tool detects five deadlocks that are missed by previous deadlock detection tools in Apache, MySql and Mozilla, and, moreover, *two of the five are new bugs and have never been reported before*. In addition, even though some(16) of the detected issues are not deadlocks, they are still bad practices and may introduce some performance issues or future deadlocks. The synchronization waiting loop inside a critical section protected by locks can potentially cause cascading wait effects among threads.

As the second use case, we extend the Valgrind [33] data race checker to leverage the ad hoc synchronization information annotated by SyncFinder. As a result, Valgrind's false positive rates for data races decrease by 43–86%. This indicates that even though SyncFinder is not a bug detector itself, it can help concurrency bug detectors improve their accuracy by providing ad hoc synchronization information.

2 Ad Hoc Synchronization Characteristics

To understand ad hoc synchronization characteristics, we have manually studied 12 representative applications of three types (server, desktop and scientific/graphic), as shown on Table 2. Two inspectors separately investigated almost every line of source code and compared the results with each other. As shown on Table 3, in our initial study, we missed a few ad hoc synchronizations, most of which are those implemented using interlocked or nested goto loops (e.g., the example in Figure 4). Fortunately, our automatic identification tool, SyncFinder, discovers them, and we were able to extend our manual examination to include such complicated types.

Threats to Validity. Similar to previous work, characteristic studies are all subject to the validity problem. Potential threats to the validity of our characteristic study are the representativeness of applications and our examination methodology. To address the former, we chose a variety of concurrent programs, including four servers, three client/desktop concurrent applications as well as five scientific applications from SPLASH-2, all written in C/C++, one of the popular languages for concurrent programs. These applications are well representative of server, client/desktop-based and scientific applications, three large classes of concurrent programs.

Apps.	#sync loops	I_a	I_b	both
Apache	33	4	2	2
MySQL	83	12	8	7
OpenLDAP	15	3	3	2
PBZip2	7	1	0	0

Table 3: **Ad hoc sync loops missed by human inspections.** Two inspectors, I_a and I_b, investigate the same source code separately. Most of the sync loops missed by both inspectors (i.e., those in Apache and MySQL) are interlocked or nested goto loops. Others (in OpenLDAP) are for-loops doing complicated useful work and checking synchronization condition in it, like one in Figure 1(d).

In terms of our examination methodology, we have examined almost every line of code including programmers' comments. This was an immensely time consuming effort that took three months of our time. To ensure correctness, the process was repeated twice, each time by a different author. Furthermore, we were also quite familiar with the examined applications, since we have modified and used them in many of our previously published studies.

Overall, while we cannot draw any general conclusions that can be applied to all concurrent programs, we believe that our study does capture the characteristics of synchronizations in three large important classes of concurrent applications written in C/C++.

Finding 1: Every studied application uses ad hoc synchronizations. More specifically, there are 6–83 ad hoc synchronizations in each of the 12 studied programs. As shown in Table 2, ad hoc synchronizations are used in all of our evaluated programs, and some programs (e.g. MySQL) even use as many as 83 ad hoc synchronizations. This indicates that, in the real world, it is not rare for programmers to use ad hoc synchronizations in their concurrent programs.

While we are not 100% sure why programmers use ad hoc synchronizations, after studying the code and comments, we speculate there are two primary reasons. The first is because there are diverse synchronization needs to ensure execution order among threads. Unlike atomicity synchronization that shares a common goal, the exact synchronization scenario for order ensurance may vary from one to another, making it hard to design a common interface to fit every need (more discussion in Finding 2).

The second reason is due to performance concerns on synchronization primitives, especially those heavyweight ones implemented as system calls. If the synchronization condition can be satisfied quickly, there is no need to pay the high overhead of context switches and system calls.

Apps.	Total loops	Total Ad hoc	Single exit condition					Multiple exit cond.			Total func	async
			sc -dir	sc -df	sc -cf	sc -func	total	mc -all	mc -Nall	total		
Apache	1462	33	4	0	1	3	8	22	3	25	16	25
MySQL	4265	83	23	5	4	11	43	13	27	40	32	64
OpenLDAP	2044	15	2	0	0	2	4	4	7	11	9	15
Cherokee	748	6	0	2	0	1	3	0	3	3	1	5
Mozilla-js	848	17	2	4	1	4	10	4	1	5	5	15
PBZip2	45	7	0	0	0	1	1	0	6	6	7	7
Transmission	1114	13	6	0	0	1	7	0	6	6	3	2
Radiosity	80	12	5	5	1	0	11	1	0	1	0	1
Barnes	88	7	6	1	0	0	7	0	0	0	0	0
Water	84	9	9	0	0	0	9	0	0	0	0	0
Ocean	339	20	20	0	0	0	20	0	0	0	0	0
FFT	57	7	7	0	0	0	7	0	0	0	0	0

sc : single exit cond.
-dir: directly depends on a sync var
-df : has data dependency
-cf : has control dependency

mc : multiple exit cond.
-all: all exit conditions depend on sync vars
-Nall : not all, but at least one does

func : inter-procedural dependency
async: useful work while waiting

Table 4: **Diverse ad hoc synchronizations in concurrent programs we studied.** (i) The number of exit conditions in synchronization loops are various (*sc vs. mc*); (ii) There can be multiple, different types of dependency relations between sync variables and loop exit conditions (*-dir, -df, -cf, -func*); (iii) Some synchronization loops do useful work with asynchronous condition checking (*async*).

```
while(1) {
    int oldcount = (global->barrier).count;
    ...
    If(updatedcount == oldcount)  break;
}                                  /* SPLASH2 */
```

(a) sc-df (data dependency)

```
int finished = 0;
for(i = 0; i < 1000 && ! finished; i ++) {
      if(global->pbar_count >= n_proc)
        finished = 1;
}                                  /* Radiosity*/
```

(b) mc-Nall (some are local exit conditions)

```
/* wait for the next block from
   a producer queue */
safe_mutex_lock(fifo->mut);
for(;;) {
    If( !queue->empty &&
        queue->getData(Data) )
        break;
}
```

```
Bool queue::getData(ElemPtr &fileData) {
    ElemPtr &headElem = qData[head];
    ...
  /* search qData to find the requested
     block.  If finds out, return true;
     otherwise, return false  */
}                                  /* PBZip2 */
```

(c) Function call

Figure 5: **Examples of various ad hoc synchronizations.** A sync variable is highlighted using a bold font. An arrow shows the dependency relation from a sync variable to a loop-exit condition. The examples of other ad hoc categories are shown on Figure 1.

Such performance justifications are frequently mentioned in programmers' comments associated with ad hoc synchronization implementations.

While ad hoc synchronizations are seemly justified, are they really worthwhile? What are their impact on program correctness and interaction with other tools? Can they be expressed using some common, easy-to-recognize synchronization primitives? We will dive into these questions in our finding 3 and 4, trying to shed some lights into the tradeoffs.

Finding 2: Ad hoc synchronization is diverse.
Table 4 further categorizes ad hoc synchronizations from several perspectives. Some real world examples for each category can be found in Figure 1 and Figure 5.

(i) Single vs. multiple exit conditions: Some ad hoc synchronization loops have only one exit condition [1]. We call such sync loops *sc* loops. Unfortunately, many others (up to 86% of ad hoc synchronizations in a program) have more than one exit condition. We refer to them as *mc* loops. In some of them (referred to as *mc_all*), all exit conditions are satisfied by remote threads. In the other loops (referred to as *mc_Nall*), there are also some *local* exit conditions such as time-outs, etc., that are independent of remote threads and can be satisfied locally.

(ii) Dependency on sync variables: The simplest ad hoc synchronization is just directly spinning on a sync variable as shown on Figure 1(a). In many other cases (50-100% of ad hoc synchronizations in a program), exit conditions indirectly depend on sync variables via data dependencies (referred to as *df*, Figure 5(a)), control dependencies (referred to as *cf*, (Figure 1(c)), even inter-procedural dependencies (referred to as *func*, Figure 5(c)).

(iii) Asynchronous synchronizations (referred as async): In some cases (77% of ad hoc synchronizations in server/desktop applications we studied), a thread does not just wait in synchronization. Instead, it also performs some useful computations while repetitively checking sync variables at every iteration. For example, in Figure 1(d), a MySQL master thread does background tasks like log flushing until a new SQL query arrives (by checking *new_activity_counter*).

[1] A condition that can break the execution out of a loop.

```
/* get tuple Id of a table */
do {
  ret= m_skip_auto_increment ?
       readAutoIncrementValue(...):
       getAutoIncrementValue(•••);
} while(ret== -1 && --retries && ..);
                              /* MySQL */
```

```
for (;;) {
  if (m_skip_auto_increment &&
      readAutoIncrementValue(...)
      || getAutoIncrementValue(...){
      if (--retries && ...) {
          my_sleep(retry_sleep);
          continue;     /* 30 ms sleep
      }                    for transaction */
    } break;
}
```

Figure 6: **An ad hoc synchronization in MySQL was revised by programmers to solve a performance problem.**

Comment examples
Programmers are aware of better design but still use ad hoc implementation (8%)
/* This can be built in *smarter way, like pthread_cond*, **but we do it** since the status can come from.. */ /* By doing.. applications will get *better performance and avoid the problem entirely*. **Regardless, we do this**... because we'd rather write error message in this routine, ..*/
Programmers try to prevent bugs at the first place (22%)
/* We could end up **spinning indefinitely** with a situation where.. The 'i++' stops the infinite loop */ /* We can safely wait here in the case.. without fear of **deadlock** because we made.. */ /* This spinning actually isn't necessary except when the **compiler does corrupt** 64bit arithmetic.. */
Programmers explicitly state their sync assumptions (75%)
/* GC doesn't set the flag until it has waited for **all active requests to end** */ /* We must break the wait **if one of the following occurs**: i).. ii).. iii).. iv).. v).. */

Table 5: **Observations in programmers' comments on ad hoc synchronization from Apache, Mozilla, and MySQL**. We study 63 comments associated with ad hoc synchronizations.

Finding 3: Ad hoc synchronizations can easily introduce bugs or performance issues.
After studying the 5 applications listed in Table 1, we found that 22–67% of synchronization loops previously introduced bugs or performance issues. These high issue rates are alarming, and, as a whole, may be a strong sign that programmers should stay away from ad hoc synchronizations.

For each ad hoc synchronization loop, we use its corresponding file and function names to find out in the source code repository if there was any patch associated with it. If there is, we manually check if the patch involves the ad hoc sync loop. We then uses this patch's information to search the bugzilla databases and commit logs to find all relevant information. By examining such information as well as the patch code, we identify whether the patch is a feature addition, a bug not related to synchronization, or a bug caused exactly by the ad hoc sync loop. We only count the last case.

Besides deadlocks (as demonstrated in Figure 2 and 3), ad hoc synchronization can also introduce other types of concurrency bugs. In some cases, an ad hoc synchronization fails to guarantee an expected order and lead to a crash because the exit condition can be satisfied by a third thread unexpectedly. Due to space limitations, we do not show those examples here.

In addition to bugs, ad hoc synchronizations can also introduce performance issues. Figure 6 shows such an example. In this case, the busy wait can waste CPU cycles and decrease throughput. Therefore, programmers revised the synchronization by adding a sleep inside the loop.

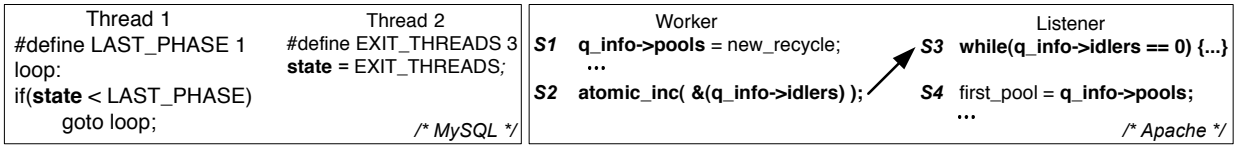
Ad hoc synchronizations also have problematic interactions with modern hardware's relaxed consistency models [5, 28, 45]. These modern microprocessors can reorder two writes to different locations, making ad hoc synchronizations such as the one in Figure 1(a) fail to guarantee the intended order in some cases. As such, experts recommended programmers to stay away from such ad hoc synchronization implementations, or at least implement synchronizations using atomic instructions instead of just simple reads or writes [5, 28, 45].

To make things even worse, ad hoc synchronizations also have problematic interactions with compiler optimizations such as loop invariant hoisting. Programmers should avoid such optimizations on sync variables, and ensure that waiting loops always read the up-to-date values instead of the cached values from registers. As a workaround, programmers may need to use wrapping variable accesses with function calls [3]. All of these just complicate programming as well as software testing and debugging.

Interestingly, some programmers are aware of the above ad hoc synchronization problems but still use them. We study the 63 comments associated with ad hoc synchronizations in MySQL, Apache, and Mozilla. As illustrated in Table 5, programmers sometimes mentioned better alternatives, but they still chose to use their ad hoc implementations for *flexibility*. In some cases, they explicitly indicated their preference for the *lightness and simplicity* of ad hoc spinning loops, especially when the synchronizations were expected to rarely occur or rarely need to wait long. Also, programmers often explicitly stated their assumptions/expectation in comments about what remote threads should do correspondingly, since ad hoc synchronizations are complex and hard to understand.

Finding 4: Ad hoc synchronizations can significantly impact the effectiveness and accuracy of concurrency bug detection and performance profiling tools.
As mentioned earlier, since existing concurrency bug (deadlock, data race) detection tools cannot recognize ad hoc synchronizations, they will fail to detect bugs that involve such synchronizations (e.g. deadlock examples shown on Figure 2 and 3).

In addition, they can also introduce many false positives. It has been well known that most data race detectors incur high false positives due to ad hoc synchronizations. Such false positives come from two sources: (1) *Benign*



(a) Benign data race on **state**

(b) False data race on **q_info->pools**

Figure 7: **False positives in Valgrind data race detection due to ad hoc synchronizations.**

data races on sync variables: typically an ad hoc synchronization is implemented via an intended data race on sync variables. Figure 7(a) shows such a benign data race reported by Valgrind [33] in MySQL. (2) *False data races that would never execute in parallel due to the execution order guaranteed by ad hoc synchronizations*: For example, in Figure 7(b), the two threads are synchronized at S2 and S3, which guarantees the correct order between S1 and S4's accesses to *q_info→pools*. S1 and S4 would never race with each other. However, most data race checkers cannot recognize this ad hoc synchronization and, as a result, incorrectly report S1 and S4 as a data race.

Synchronization is also a big performance and scalability concern because time waiting at synchronization is wasted. Unfortunately, existing work in synchronization cost analysis [25, 32] and performance profiling [29] cannot recognize ad hoc synchronizations, and therefore the synchronizations can easily be mistaken as computation. As a result, the final performance profiling results may cause programmers to make less optimal or even incorrect decisions while performance tuning.

Replacing with synchronization primitives. Our findings above reveal that ad hoc synchronization is often harmful in several respects. Therefore, it is desirable that programmers use synchronization primitives such as cond_wait, rather than ad hoc synchronization. Figure 8 shows how ad hoc synchronization can be replaced with a well-known synchronization primitive, POSIX pthread_cond_wait(). Note that it may not always be straightforward to use existing synchronization primitives to replace all ad hoc synchronizations, because existing synchronization primitives may not be sufficient to meet the diverse synchronization needs as well as the performance requirements, as discussed in Finding 1.

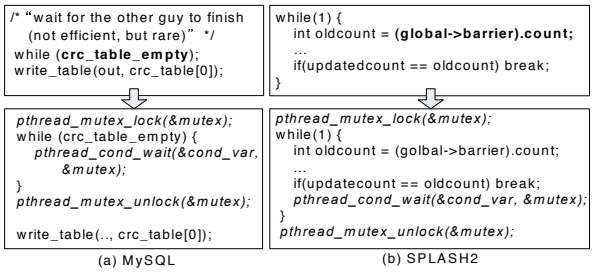


Figure 8: **Replacing ad hoc synchronizations with synchronization primitives using condition variables.** (a) shows the re-implementation of ad hoc synchronization in Figure 1(a); (b) is for Figure 5(a).

3 Ad hoc Synchronization Identification

3.1 Overview

As ad hoc synchronizations have raised many challenges and issues related to correctness and performance, it would be useful to identify and annotate them. Manually doing this is tedious and error-prone since they are diverse and hard to tell apart from computation. Therefore, the second part of our work builds a tool called SyncFinder to automatically identify and annotate them in the source code of concurrent programs. The annotation can be leveraged in several ways as discussed in Section 1.2.

There are two possible approaches to achieve the above goal. One is dynamic and is done by analyzing run-time traces. The other approach is static, involving the analysis of source code. Even though the dynamic approach has more accurate information than the static method, it can incur large (up to 30X [27]) run-time overhead to collect memory access traces. In addition, the number of ad hoc synchronizations that can be identified using this method would largely depend on the code coverage of test cases. Also some ad hoc synchronization loops may terminate after only one iteration, making it hard to identify them as ad hoc synchronization loops [18]. Due to these reasons, we choose the static method, i.e., analyzing source code.

The biggest challenge to automatically identify ad hoc synchronizations is how to separate them from computation loops. The diversity of ad hoc synchronizations makes it especially hard. To address the above challenge, we have to identify the common elements among various ad hoc synchronization implementations.

Commonality among ad hoc synchronizations: Interestingly, ad hoc synchronizations are all implemented using loops, referred to as *sync loops* (Figure 9). While a sync loop can have many exit conditions, at least one of them is the exit condition to be satisfied when an expected synchronization event happens. We refer to such exit conditions as *sync conditions*. The sync condition directly or indirectly depends on a certain shared variable (referred as *a sync variable*) that is loop-invariant locally, and modified by a remote thread.

Note that a sync variable may not necessarily be directly used by a sync condition (e.g., inside a while loop condi-

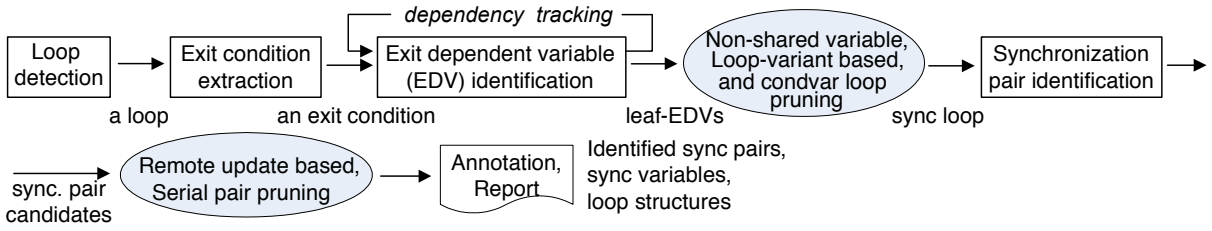


Figure 10: **SyncFinder design to automatically identify and annotate ad hoc synchronization**

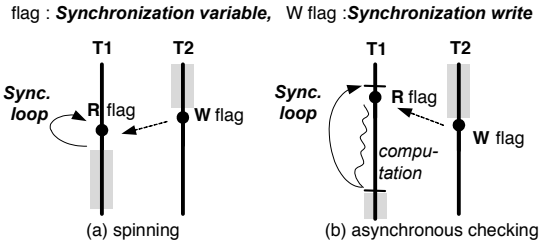


Figure 9: **Ad hoc synchronization abstract model.** The loop exit condition (i.e., sync condition) either directly or indirectly depends on a sync variable.

tion). Instead, a sync condition may have data/control-dependency on it like in the examples shown on Figure 1(c) and Figure 5(a)(c).

Following the above characteristic, SyncFinder starts from loops in the target programs, and examines their exit conditions to identify those that are (1) loop invariant, (2) directly or indirectly depend on a shared variable, and (3) can be satisfied by a remote thread's update to this variable. By checking these constraints, SyncFinder filters out most computation loops as shown in our evaluation.

Checking all of the above conditions requires SyncFinder to conduct (1) program analysis to know the exit conditions for each loop; (2) data and control flow analysis to know the dependencies of exit conditions; (3) some static thread analysis to conservatively identify what segment of code may run concurrently; and (4) some simple satisfiability analysis to check whether the remote update to the sync variable can satisfy the sync condition.

As shown on Figure 10, SyncFinder consists of the following steps: (1) Loop detection and exit condition extraction; (2) Exit dependent variable (EDV) identification; (3) Pruning computation and condvar loops based on characteristics of EDVs; (4) Synchronization pairing to pair an identified sync loop with a remote update that would break the program out of this sync loop; (5) Final result reporting and annotation in the target program's source code.

SyncFinder is built on top of the LLVM compiler infrastructure [23] since it provides several useful basic features that SyncFinder needs. LLVM's intermediate representation (IR) is based on single static assignment (SSA) form, which automatically provides a compact definition-use graph and control flow graph for every function, both of which can be leveraged by SyncFinder's data-, and control-flow analysis. In addition, SyncFinder also uses LLVM's loopinfo analysis, alias analysis, and constant propagation tracking to implement the ad hoc sync loop identification algorithm. SyncFinder annotation is done via the static instrumentation interfaces provided in LLVM. In the rest of this section, we focus on our algorithms and do not go into details about the basic analysis provided by LLVM.

3.2 Finding Loops

Apps.	while	for	goto	Total
Apache	27	4	2	33
MySQL	33	24	26	83
OpenLDAP	7	4	4	15
Mozilla-js	12	4	1	17

Table 6: **Loop mechanisms used for real-world ad hoc synchronization.** There are a non-negligible number of "goto" loops, which often complicate loop analysis (e.g., Figure 4).

As shown in Table 6, ad hoc synchronizations are implemented using three primary forms of loops: "while", "for" and "goto". Fortunately, LLVM's loopinfo pass identifies all those loops based on back edges in LLVM IR.

For each loop identified by LLVM, SyncFinder extracts its exit conditions. Specifically, it identifies the basic blocks with at least one successor outside of the loop, then for each identified basic block, SyncFinder extracts its terminator instruction, from which SyncFinder can identify the branch conditions. Such conditions are the exit conditions for this loop. SyncFinder represents the exit conditions in a canonical form: disjunction (OR) of multiple conditions, and examines each separately.

In addition, since LLVM does not keep the loop context information, e.g., loop headers and bodies, across functions, SyncFinder keeps track of them into its own data structure and uses them throughout the analysis.

3.3 Identifying Sync Loops

The key challenge of SyncFinder is to differentiate sync loops from computation loops. To address this challenge, SyncFinder examines the exit conditions of each loop by going through the following steps to filter out computation loops.

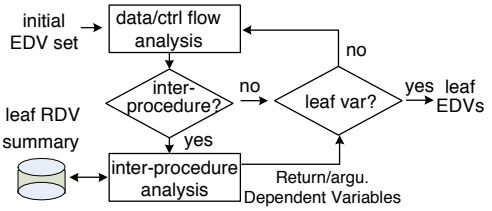


Figure 11: **Leaf-EDV identification.** SyncFinder recursively tracks Exit Dependent Variables(EDVs) along the data-, control-flow, until it reaches a leaf-EDV.

(1) Exit Dependent Variable (EDV) analysis : For each exit condition of each loop in the target program, the first step is to identify all variables that this exit condition depends on—we refer to them as exit dependent variables (EDVs). If a loop is a sync loop, the sync variables should be included in its EDVs. Note that a sync variable is not necessarily used in an exit condition (sync condition) directly. A loop exit condition can be data/control-dependent on a sync variable. Therefore, we conduct data-flow and control-flow analysis to find indirect EDVs. The EDV identification process is similar to static backward slicing [48, 38, 15].

SyncFinder first starts from variables directly referenced in the exit condition. They are added into an EDV set. Then, as shown in Figure 11, it pops a variable out from the EDV set, and finds out new EDVs along this variable's data/control flow. New EDVs are inserted into the set. It then pops another EDV from the set, and so on so forth until it reaches the loop boundary. For an EDV that does not depend on any other variables inside this loop, we refer them as a *leaf-EDV* (similar to "live-in" variables). SyncFinder maintains a separate set for leaf-EDVs. Obviously, leaf-EDVs are the ones we should focus on since they are not derived from any other EDVs in this loop.

During the backward data/control flow tracking process, if the dependency analysis encounters a function whose return value or passed-by-reference arguments affect the loop exit condition, SyncFinder further tracks the dependency via inter-procedural analysis. SyncFinder applies data- and control-flow analysis starting from the function's return value, and identifies Return/arguments-Dependent Variables (RDVs) in the callee. Such RDVs are also added into the leaf-EDV set. In addition, all RDVs of this function are stored in a summary to avoid analyzing this function again for other loops.

To handle variable and function pointer aliasing, SyncFinder leverages and extends LLVM's alias analysis to allow it go beyond function boundary.

(2) Pruning computation loops For *every* exit condition of a loop, SyncFinder applies the following two pruning steps to check whether it is a sync condition. At the end, if a loop has *at least* one sync condition, it is identified as a sync loop. Otherwise, it is pruned out as a computation loop. Most computation loops are filtered in this phase.

Non-shared variable pruning: A sync variable should be a shared variable that can be set by a remote thread. Specifically, it should be either a global variable, a heap object, or a data object (even stack-based) that is passed to a function (e.g., thread starter function) called by another thread, which can be shared by the two threads.

Therefore, if an exit condition has no shared variables in its leaf-EDV set, it is deleted from the loop's exit condition set. SyncFinder moves to the next exit condition of this loop. If the loop has no exit conditions left, this loop is pruned out as a computation loop.

Loop-variant based pruning: In almost all cases, a sync condition is loop-invariant locally, and only a remote thread changes the result of the sync condition. Based on this observation, SyncFinder prunes out those exit conditions that are loop-variant locally as shown on Figure 12. It is possible that some ad hoc synchronizations may also change the sync conditions locally. In all our experiments with 25 concurrent programs, we did not find any true ad hoc synchronizations that SyncFinder missed due to this pruner. Note that some exit conditions, such as expiration time, are separated as different conditions, and we examine *each* condition *separately*.

```
while(module){
  next = module->next;
  free(module);
  module = next;    } /* Mozilla */
```
(a) Loop-variant *module*

```
for (i = 0; i < nlights; i++){
  VecMatMult(lp->pos, m, lp->pos);
  lp = lp->next;
}                              /* SPLASH */
```
(b) Loop-variant condition checking

Figure 12: **The non-sync variables pruned out by loop-variant based pruning**. In the two computation loops, the variables in italic font are shared variable leaf-EDVs.

To check if an exit condition is loop variant, SyncFinder applies a modification (MOD) analysis within the scope of a loop being examined. Specifically, it checks all leaf-EDVs and leaf-RDVs of this loop, and prunes out those modified locally within this loop. The leaf-RDV summary is also updated accordingly.

(3) Pruning condvar loops: SyncFinder does not consider condvar loops (i.e., sync loops that are associated with cond_wait primitives) as ad hoc loops as they can be easily recognized by intercepting or instrumenting these primitives. As the final step of the ad hoc sync loop identification, SyncFinder checks every loop candidate to see it calls a cond_wait primitive inside the loop. Loops that use primitives are recognized as condvar loops and are thereby pruned out. The names of cond_wait primitives(original pthread functions or wrappers) are provided as input to SyncFinder to identify cond_wait calls.

3.4 Synchronization Pairing

Once we identify a potential sync loop, we find the remote update (referred as *a sync write*) that would "release" (break) the wait loop. To identify a sync write, SyncFinder

Apps.	total	constant	inc/dec op
Apache	42	21 (50.0%)	5 (11.9%)
MySQL	325	125 (38.5%)	110 (33.8%)
OpenLDAP	203	48 (23.6%)	8 (3.9%)
Mozilla-js	83	41 (49.4%)	31 (37.3%)

Table 7: **The characteristics of writes to sync variables.** In the four sampled applications, majority of writes assign constant values, or use simple increase or decrease operations.

first collects all write instructions modifying sync variable candidates, and then applies the following pruning steps.

Pruning unsatisfiable remote updates For each remote update to the target sync variable candidate, SyncFinder analyzes what value is assigned to this variable, and whether it can satisfy the sync condition. A complicated solution to achieve this functionality is to use a SAT solver. But it is too heavyweight, especially since, according to our observations (shown in Table 7), the majority(66%) of sync writes either assign constant values to sync variables, or use simple counting operations like increment/decrement, rather than complicated computations. This is because a sync variable is usually a control variable (e.g. status, flag, etc.) and does not require sophisticated computations.

Therefore, instead of using a SAT solver, we use constant propagation to check if this remote update would satisfy the exit condition. For an assignment with a constant, it substitutes the variable with the constant, and propagates it till the exit condition to see if it is satisfiable or not. For increment based updates, SyncFinder treats it as "sync var > 0" since it obviously does not release the loop that is waiting for an exit condition "(sync var == 0)".

Pruning serial pairs A sync loop and a sync write should be able to execute concurrently. If there is a happens-before relation between such pair, due to thread creation/join, barrier, etc, the remote write does not match with the sync loop. Due to the limitation of static analysis, currently SyncFinder conservatively prunes serial pairs related to only thread creation/join. Specifically, SyncFinder follows thread creation and conservatively estimates code that might be running concurrently.

3.5 SyncFinder Annotation

After the above pruning process, the remaining ones are identified as sync loops, along with their corresponding sync writes. All the results are stored in a file. SyncFinder also automatically annotates in the target software's source code using LLVM static instrumentation framework. It inserts `//#SyncAnnotation: Sync_Loop_Begin(&loopId)`, `//#SyncAnnotation: Sync_Loop_End(&loopId)`, respectively, at the beginning and end of an identified sync loop. In addition, inside the loop, it also annotates the read to a sync variable by inserting `//#SyncAnnotation: Sync_Read(&syncVar, &loopId)`. For the corresponding sync write, it inserts `//#SyncAnnotation: Sync_Write(&syncVar, &loopId)`. The loopId is used to match a remote sync write with a sync loop. Similar annotations are also inserted into the target program's bytecode to be leveraged by concurrency bug detection tools as discussed in the next section.

4 Two Use Cases of SyncFinder

SyncFinder's auto-identification can be used by many bug detection tools, performance profiling tools, concurrency testing frameworks, program language designers, etc. We built two use cases to demonstrate its benefits.

4.1 A Tool to detect bad practices

It is considered bad practice to wait inside a critical section, as it can easily introduce deadlocks like the Apache example shown on Figure 2 and the MySQL example on Figure 3. Furthermore, it can result in performance issues caused by cascading wait effects, and may introduce deadlocks in the future if programmers are not careful. As a demonstration, we built a simple detector (referred to as *wait-inside-critical-section detector*) to catch these cases leveraging SyncFinder's auto-annotation of ad hoc synchronizations. Our detection algorithm can be easily integrated into any existing deadlock detection tool as well.

To detect such pattern, our simple detector checks every sync loop annotated by SyncFinder to see if it is performed while holding some locks. If a sync loop is holding a lock, then SyncFinder checks the remote sync write to see whether the write is performed after acquiring the same lock or after another ad hoc sync loop, so on and so forth, to see if it is possible to form a circle. If it is, the detector reports it as a potential issue: either a deadlock or at least a bad practice.

4.2 Extensions to data race detection

We also extend Valgrind [33]'s dynamic data race detector to leverage SyncFinder's auto-identification of ad hoc sync loops. Valgrind implements a happens-before algorithm [21] using logical timestamps, which was originally based on conventional primitives including mostly lock primitives, and thread creation/join. It *cannot* recognize ad hoc synchronizations. As a result, it can introduce many false positives (shown in Table 12) as discussed in Section 2 and illustrated using two examples in Figure 7.

We extend Valgrind to eliminate data race false positives by considering ad hoc synchronizations annotated by SyncFinder. It treats the end of a sync loop in a similar way to a cond_wait operation, and the corresponding sync write like a signal operation. This way it keeps track of the happens-before relationship between them. We also extend Valgrind to not consider sync variable reads and writes as data races.

	Apps.	Total loops	Identified Sync Loops			Missed ones
			Total	True	FP	
Server	Apache	1462	17	15	2	1
	MySQL	4265	48	42	6	3
	OpenLDAP	2044	18	14	4	1
	Cherokee	748	6	6	0	0
	AOLServer	496	6	6	0	-
	Nginx	705	12	11	1	-
	BerkeleyDB	1006	15	11	4	-
	BIND9	1372	5	4	1	-
Desktop	Mozilla-js	848	16	11	5	1
	PBZip2	45	7	7	0	0
	Transmission	1114	14	12	2	1
	HandBrake	551	13	13	0	-
	p7zip	1594	10	9	1	-
	wxDFast	154	6	6	0	-
Scientific	Radiosity	80	12	12	0	0
	Barnes	88	7	7	0	0
	Water	84	9	9	0	0
	Ocean	339	20	20	0	0
	FFT	57	7	7	0	0
	Cholesky	362	8	8	0	-
	RayTracer	144	3	3	0	-
	FMM	108	8	8	0	-
	Volrend	77	9	9	0	-
	LU	38	0	0	0	-
	Radix	52	14	14	0	-
	Total (Ave.)	-	290 (11.6)	264 (10.6)	26 (1.0)	-

Table 8: **Overall results of SyncFinder:** *Every concurrent program uses ad hoc sync loops except LU*. Both true ad hoc sync loops and false positives are showed here. For the 12 programs used in the characteristic study, the numbers of missed ad hoc sync loops are also reported. They are generated by comparing with our manual checking results from the characteristic study. We cannot show the numbers of missed ad hoc sync loops for the unseen programs in the study since we did not manually examine them as we did for the 12 studied programs. To show SyncFinder's total exploration space, we also show the total number of loops, most of which are computation loops. Note that the total numbers of ad hoc sync loops are different from those numbers shown in Table 2 because some code (for other platforms such as FreeBSD, etc) are not included during the compilation.

5 Evaluation

5.1 Effectiveness and Accuracy

We evaluated SyncFinder on 25 concurrent programs, including 12 used in our manually characteristic study and 13 other ones. Table 8 shows the overall result of SyncFinder on the 25 programs. On average SyncFinder accurately identifies 96% of ad hoc sync loops in the 12 studied programs and has a 6% false positive rate overall. SyncFinder successfully identified diverse ad hoc order synchronizations, including those we missed during our manual identification. For example, it successfully identifies those complicated, interlocked "goto" sync loops, as shown in Figure 4.

For the 12 studied programs, SyncFinder misses a few(1-3 per application) sync loops in large server/desktop applications. Considering the total number of loops (up to 4265) in each of these applications, such a small miss rate does not limit SyncFinder's applicability to real world programs. SyncFinder fails to identify these sync loops because of the unavailability of the source code for these library functions and inaccurate pointer alias.

SyncFinder also returns a low number of false positives for all 25 programs. As showed in Table 8, SyncFinder has 0-6 false positives per program (i.e. a false positive rate of 0-30%). Such numbers are quite reasonable. Programmers can easily examine the reported sync loops to prune out those few false positives. Most of the false positives are caused by inaccurate function pointer analysis. Due to complicated function pointer alias, sometimes SyncFinder cannot further track into callee functions to check if a target variable (leaf-EDV) is locally modified. In these cases, SyncFinder conservatively considers the target variable as a sync variable.

5.2 Sync Loop Identification and Pruning

Apps.	Total loops	Exit cond.	Leaf-EDVs	Aft non--shared pr.	Aft loop-var. pr.	Aft cond-var pr.
Apache	1,462	3,120	8,682	184	24	20
MySQL	4,265	9,181	20,458	377	118	72
OpenLDAP	2,044	4,434	11,276	171	45	27
PBZip2	45	278	799	130	16	9

Table 9: **EDV Analysis and non-sync variable pruning**. After identifying leaf-EDVs for each loop, SyncFinder applies non-shared, loop-variant and condvar-loop based pruning schemes. The final results are the sync variables of the ad hoc sync loops. Some sync variables may be associated with a same sync loop.

To show the effectiveness of sync loop identification, in Table 9, we test SyncFinder on some server/desktop applications and show the results from each of the sync loop identification steps. From the total loops identified, SyncFinder extracts exit conditions, and identifies all leaf-EDVs (the third column in Table 9). From the leaf-EDVs, SyncFinder prunes out non-shared variables (95% of leaf-EDVs), and applies loop-variant based pruning, which further prunes 80% of shared leaf-EDVs. SyncFinder then applies the final pruning step to prune out sync variables that are associated with condvar loops. The remains are sync variable candidates and those loops using them are potential sync loops.

5.3 Synchronization Pairing and Pruning

During synchronization pairing, SyncFinder applies two pruning schemes, unsatisfiable remote update pruning and

Apps.	Initial pairs	w/ Remote update pr.	w/ Serial pair pr.	With both	True pairs
Apache	27	22	27	22	21
MySQL	251	204	178	141	123
OpenLDAP	168	134	146	115	96
PBZip2	19	15	11	9	9

Table 10: **False synchronization pair pruning**. Note that the numbers shown here are synchronization *pairs*. In all the other results, we show "synchronization loops" (regardless how many setting statements for an ad hoc sync loop)

serial pair pruning. Table 10 shows the effect of those pruning steps on the same set of server/desktop applications in Table 9. First, remote update based pruning eliminates 51.8% of false sync pair candidates on average. It is especially effective on Apache, since the majority of sync writes are just simple assignments with constant values, so it is easy to determine whether such values would satisfy the corresponding sync exit conditions.

Second, the effectiveness of serial pair pruning depends on application characteristics. While it prunes out almost all false positives in simple desktop/scientific programs (e.g., PBZip2), it is less effective in servers like Apache, where many function pointers are used. Due to the limitation of function pointer analysis, it is hard to know in all cases whether two certain regions cannot be concurrent. To be conservative, SyncFinder does not prune the pairs inside such regions. Fortunately, the remote update based pruning helps filtering them out.

5.4 Two Use Cases: Bug Detection

Apps.	Deadlock (New)	Bad practice
Apache	1 (0)	1
MySQL	2 (2)	13
Mozilla	2 (0)	2

Table 11: **Deadlock and bad practice detection**

Table 11 shows that our *simple* deadlock detector (leveraging SyncFinder's ad hoc synchronization annotation) detects five deadlocks involving ad hoc order synchronizations, including those shown in Figure 2 and Figure 3. Previous tools would fail to detect these bugs since they cannot recognize ad hoc synchronizations. Besides deadlocks, our detector also reports 16 bad practices, i.e. waiting in a sync loop while holding a lock, which could raise performance issues or cause future deadlocks.

Apps.	Original Valgrind	Extended Valgrind	%Pruned
Apache	30	17	43%
MySQL	25	10	60%
OpenLDAP	7	4	43%
Water	79	11	86%

Table 12: **False positive reduction in Valgrind**

Table 12 shows that SyncFinder auto-annotation could reduce the false positive rates of Valgrind data race detector by 43-86%.

6 Related Work

Spin and hang detection Some recent work has been proposed in detecting *simple* spinning-based synchronizations [32, 25, 18]. For example, [25] proposed some new hardware buffers to detect spinning loops on-the-fly. [18] also provides similar capability but does it in software. Both can detect only simple spinning loops, i.e. those sync loops with only one single exit condition and also directly depend on sync variables (referred as "sc-dir" in Table 3 in Section 2). As shown in Table 3 such simple spinning loops account for less than 16% of ad hoc sync loops on average in server/desktop applications we studied.

Besides, both of them are dynamic approaches and thereby suffer from the coverage limitation of all dynamic approaches (discussed in Section 3). In contrast, SyncFinder uses a static approach and can detect various types of ad hoc synchronizations. Additionally, we also conduct an ad hoc synchronization characteristic study.

Synchronization annotation Many annotation languages [4, 2, 1, 41] have been proposed for synchronizations in concurrent programs. Unfortunately, annotation is not frequently used by programmers since it is tedious. SyncFinder is complementary to these work by providing automatic annotation for ad hoc synchronizations.

Concurrent bug detection tools Much research has been conducted on concurrency bug detection [47, 20, 31, 6, 17, 11, 43]. These tools usually assume that they can recognize all synchronizations in target programs. As we demonstrated using deadlock detection and race detection, SyncFinder can help these tools improve their effectiveness and accuracy by automatically annotating ad hoc synchronizations that are hard for them to recognize.

Transactional memory Various transactional memory designs have been proposed to solve the programmability issues related to mutexes [39, 30, 19, 44] and also condition variables [10]. Our study complements such work by providing ad hoc synchronization characteristics in real world applications.

Software bug characteristics studies Several studies have been conducted on the characteristics of software bugs [8, 42, 34], including one of our own [26] on concurrency bug characteristics. This paper is different from those studies by focusing on ad hoc synchronizations instead of bugs, even though many of them are prone to introducing bugs. The purpose of this paper is to raise the awareness of ad hoc synchronizations, and to warn programmers to avoid them when possible. Also we developed an effective way to automatically identify those ad hoc synchronizations in large software.

7 Conclusions and Limitations

In this paper, we provided a quantitative characteristics study of ad hoc synchronization in concurrent programs and built a tool called SyncFinder to automatically identify

and annotate them. By examining 229 ad hoc synchronization loops from 12 concurrent programs, we have found several interesting and alarming characteristics. Among them, the most important results include: all concurrent programs have used ad hoc synchronizations and their implementations are very diverse and hard to recognize manually. Moreover, a large percentage (22-67%) of ad hoc loops in these applications have introduced *bugs or performance issues*. They also greatly impact the accuracy and effectiveness of bug detection and performance profiling tools. In an effort to detect these ad hoc synchronizations, we developed SyncFinder, a tool that successfully identifies 96% of ad hoc synchronization loops with a 6% false positive rate. SyncFinder helps detect deadlocks missed by conventional deadlock detection and also reduce data race detector's false positives. Many other tools and research projects can also benefit from SyncFinder. For example, concurrency testing tools (e.g., CHESS [31]) can leverage SyncFinder's auto-annotation to force a context switch inside an ad hoc sync loop to expose concurrency bugs. Similarly, performance tools can be extended to profile ad hoc synchronization behavior.

All work has limitations, and ours is no exception: (i) SyncFinder requires source code. However, this may not significantly limit SyncFinder's applicability since it is more likely to be used by programmers instead of end users. (ii) Due to some implementation issues, SyncFinder still misses 1-3 ad hoc synchronizations. Eliminating them would require further enhancement to some of our analysis (such as alias analysis, etc.) (iii) Even though SyncFinder's false positive rates are quite low, for some use cases that are sensitive to false positives, programmers would need to manually examine the identified ad hoc synchronization or leverage some execution synthesis tools like ESD [49] to help identify false positives. (iv) For our characteristic study, we can always study a few more applications, especially of different types.

8 Acknowledgments

We would like to express our deepest appreciation to our shepherd, Professor George Candea, who was very responsive during our interactions with him and provided us with valuable suggestions, which have significantly improved our paper and strengthened our work. Moreover, we would also like to thank the anonymous reviewers whose comments and insightful suggestions have greatly shaped and improved our paper and have taught us many important lessons. Finally, we greatly appreciate Bob Kuhn, Matthew Frank and Paul Petersen for their continuous support and encouragement throughout the whole project, as well as their insightful feedback on the project and the paper. This work is supported by NSF-CSR 1017784, NSF CNS-0720743 grant, NSF CCF-1017804 grant, NSF CNS-1001158 (career award) and Intel Grant.

References

[1] Java annotations. `http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`.

[2] Lock_Lint - Static data race and deadlock detection tool for C. `http://developers.sun.com/sunstudio/articles/locklint.html`.

[3] Miscompiled volatile-qualified variables. `https://www.securecoding.cert.org/confluence/display/seccode/DCL17-C.+Beware+of+miscompiled+volatile-qualified+variables`.

[4] MSDN run-time library reference – SAL annotations. `http://msdn2.microsoft.com/en-us/library/ms235402.aspx`.

[5] BOEHM, H.-J., AND ADVE, S. V. Foundations of the c++ concurrency memory model. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2008), ACM, pp. 68–78.

[6] BRON, A., FARCHI, E., MAGID, Y., NIR, Y., AND UR, S. Applications of synchronization coverage. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA* (2005), ACM, pp. 206–212.

[7] BURNS, B., GRIMALDI, K., KOSTADINOV, A., BERGER, E. D., AND CORNER, M. D. Flux: A language for programming high-performance servers. In *USENIX Annual Technical Conference, General Track* (2006), USENIX, pp. 129–142.

[8] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. R. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles (18th SOSP'01)* (Banff, Alberta, Canada, Oct. 2001), ACM SIGOPS, pp. 73–88.

[9] CONDIT, J., HARREN, M., ANDERSON, Z. R., GAY, D., AND NECULA, G. C. Dependent types for low-level programming. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007* (2007), Springer, pp. 520–535.

[10] DUDNIK, P., AND M.SWIFT, M. Condition variables and transactional memory: Problem or oppertunity? In *4th ACM SIGPLAN Workshop on Transactional Computing(Transact)*.

[11] ENGLER, D., AND ASHCRAFT, K. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (Oct. 19–22 2003), pp. 237–252.

[12] FLANAGAN, C., AND FREUND, S. N. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2004), ACM, pp. 256–267.

[13] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *ISCA* (1993), pp. 289–300.

[14] HOARE, C. A. R. Monitors: an operating system structuring concept. *Communications of the ACM 17* (1974), 549–557.

[15] HORWITZ, S., REPS, T. W., AND BRINKLEY, D. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)* (Atlanta, Georgia, 1988), pp. 35–46.

[16] HOWARD, J. H. Proving monitors. *Commun. ACM 19*, 5 (1976), 273–279.

[17] INTEL CORPORATION. Intel thread checker. http://software.intel.com/en-us/articles/intel-thread-checker-documentation/.

[18] JANNESARI, A., AND TICHY, W. F. Identifying ad-hoc synchronization for enhanced race detection. In *IPDPS* (April 2010), IEEE.

[19] JAYARAM BOBBA, NEELAM GOYAL, M. H.-M. S. D. W. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *International Symposium on Computer Architecture* (June 2008), pp. 127–138.

[20] JULA, H., TRALAMAZZA, D. M., ZAMFIR, C., AND CANDEA, G. Deadlock immunity: Enabling systems to defend against deadlocks. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA* (2008), pp. 295–308.

[21] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[22] LAMPSON, B., AND REDELL, D. Experience with processes and monitors in mesa. *Communications of the ACM 23*, 2 (Feb. 1980), 105–117.

[23] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, Mar 2004).

[24] LI, T., ELLIS, C. S., LEBECK, A. R., AND SORIN, D. J. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *USENIX Annual Technical Conference, General Track* (2005), pp. 31–44.

[25] LI, T., LEBECK, A. R., AND SORIN, D. J. Spin detection hardware for improved management of multithreaded systems. *IEEE Transactions on Parallel and Distributed Systems PDS-17*, 6 (June 2006), 508–521.

[26] LU, S., PARK, S., SEO, E., AND ZHOU, Y. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems* (March 2008).

[27] LU, S., TUCEK, J., QIN, F., AND ZHOU, Y. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ACM, pp. 37–48.

[28] MANSON, J., PUGH, W., AND V.ADVE, S. The java memory model. In *POPL* (January 2005), ACM.

[29] MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R. B., KARAVANIC, K. L., KUNCHITHAPADAM, K., AND NEWHALL, T. The paradyn parallel performance measurement tool. *Special issue on performance evaluation tools for parallel and distributed computer systems 28* (November 1995), 37–46.

[30] MINH, C. C., TRAUTMANN, M., CHUNG, J., MCDONALD, A., BRONSON, N., CASPER, J., KOZYRAKIS, C., AND OLUKOTUN, K. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture* (New York, NY, USA, 2007), ACM, pp. 69–80.

[31] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing heisenbugs in concurrent programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings* (2008), USENIX Association, pp. 267–280.

[32] NAKKA, N., SAGGESE, G. P., KALBARCZYK, Z., AND IYER, R. An architectural framework for detecting process hangs/crashes. In *EDCC: EDCC, European Dependable Computing Conference* (2005).

[33] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (2007), 89–100.

[34] OSTRAND, T. J., WEYUKER, E. J., AND BELL, R. M. Predicting the location and number of faults in large software systems. *IEEE Trans. Software Eng 31*, 4 (2005), 340–355.

[35] PADIOLEAU, Y., TAN, L., AND ZHOU, Y. Listening to programmers taxonomies and characteristics of comments in operating system code. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 331–341.

[36] PARK, S., LU, S., AND ZHOU, Y. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009* (2009), ACM, pp. 25–36.

[37] RAJWAR, R., AND HILL, M. Transactional memory bibliography. http://www.cs.wisc.edu/trans-memory/biblio/.

[38] REPS, T., AND ROSAY, G. Precise interprocedural chopping. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering* (New York, NY, USA, 1995), ACM, pp. 41–52.

[39] ROSSBACH, C. J., HOFMANN, O. S., PORTER, D. E., RAMADAN, H. E., ADITYA, B., AND WITCHEL, E. Txlinux: using and managing hardware transactional memory in an operating system. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007* (2007), ACM, pp. 87–102.

[40] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Symposium on Principles of Distributed Computing* (1995), ACM Press.

[41] STERLING, N. WARLOCK - A static data race analysis tool. In *USENIX Winter Technical Conference* (1993), pp. 97–106.

[42] SULLIVAN, M., AND CHILLAREGE, R. A comparison of software defects in database management systems and operating systems. In *FTCS* (1992), pp. 475–484.

[43] SUN MICROSYSTEMS INC. Thread analyzer user's guide. http://dlc.sun.com/pdf/820-0619/820-0619.pdf.

[44] TIM HARRIS, SIMON MARLOW, S. P.-J. M. H. Composable memory transactions. In *ACM SIGPLAN symposium on Principles and practice of parallel programming* (2005).

[45] V.ADVE, S., AND GHARACHORLOO, K. Shared memory consistency models: A tutorial. In *computer* (1996), IEEE.

[46] VAZIRI, M., TIP, F., AND DOLBY, J. Associating synchronization constraints with data in an object-oriented language. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2006), ACM, pp. 334–345.

[47] WANG, Y., KELLY, T., KUDLUR, M., LAFORTUNE, S., AND MAHLKE, S. A. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA* (2008), pp. 281–294.

[48] WEISER, M. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering* (Piscataway, NJ, USA, 1981), IEEE Press, pp. 439–449.

[49] ZAMFIR, C., AND CANDEA, G. Execution synthesis: a technique for automated software debugging. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems* (New York, NY, USA, 2010), ACM, pp. 321–334.

[50] ZHOU, F., CONDIT, J., ANDERSON, Z. R., BAGRAK, I., ENNALS, R., HARREN, M., NECULA, G. C., AND BREWER, E. A. Safedrive: Safe and recoverable extensions using language-based techniques. In *OSDI* (2006), pp. 45–60.

Deterministic Process Groups in dOS

Tom Bergan Nicholas Hunt Luis Ceze Steven D. Gribble

Department of Computer Science & Engineering, University of Washington

Abstract

Current multiprocessor systems execute parallel and concurrent software nondeterministically: even when given precisely the same input, two executions of the same program may produce different output. This severely complicates debugging, testing, and automatic replication for fault-tolerance. Previous efforts to address this issue have focused primarily on record and replay, but making execution actually deterministic would address the problem at the root.

Our goals in this work are twofold: (1) to provide fully deterministic execution of arbitrary, unmodified, multithreaded programs as an OS service; and (2) to make all sources of intentional nondeterminism, such as network I/O, be explicit and controllable. To this end we propose a new OS abstraction, the *Deterministic Process Group* (DPG). All communication between threads and processes *internal* to a DPG happens deterministically, including implicit communication via shared-memory accesses, as well as communication via OS channels such as pipes, signals, and the filesystem. To deal with fundamentally nondeterministic *external* events, our abstraction includes the *shim layer*, a programmable interface that interposes on all interaction between a DPG and the external world, making determinism useful even for reactive applications.

We implemented the DPG abstraction as an extension to Linux and demonstrate its benefits with three use cases: plain deterministic execution; replicated execution; and record and replay by logging just external input. We evaluated our implementation on both parallel and reactive workloads, including Apache, Chromium, and PARSEC.

1. Introduction

Nondeterminism makes the development of parallel and concurrent software substantially more difficult. Software testers face daunting incompleteness challenges because nondeterminism leads to an exponential explosion in possible executions [27]. Developers must reason about large sets of possible behaviors and attempt to debug without precise repeatability [31, 36]. Moreover, standard techniques for fault-tolerant replication do not work when the software being replicated executes nondeterministically [38]. At the same time, the growing popularity of multicore architectures is making parallel and concurrent software more and more important.

Unfortunately, nondeterminism is pervasive; thread scheduling, memory reordering, and timing variations at the hardware level can all affect the interleaving of threads and cause a multithreaded program to produce different outputs when given the same input. We define this as *internal nondeterminism*. Internal nondeterminism is entirely hidden from the programmer and thus is undesirable. However, as we demonstrate in this paper, it is not fundamental and can be completely removed. On the other hand, events such as user input and the arrival of network packets are triggered nondeterministically by the external world. We define this as *external nondeterminism*; this kind of nondeterminism, if present, is fundamental and cannot be removed.

What we want is a software environment where internal nondeterminism is completely eliminated. What we want is more than just deterministic record and replay: multithreaded programs should always execute deterministically relative to their explicitly specified inputs. Moreover, where external nondeterminism exists, it should be made explicit and controllable.

Recent research has begun to explore ways of reducing internal nondeterminism in multithreaded programs. However, current proposals fall short in several aspects: they do not deal with nondeterministic channels other than shared-memory; they do not offer ways of making external nondeterminism explicit and controllable; they either require new hardware [14], apply to only a subset of programs [7, 29], or require recompilation [6]; and they do not support multiprocess applications.

Our goals are to completely eliminate nondeterminism where possible, including channels beyond shared-memory like pipes, signals, and the filesystem, and to make all intentional, external nondeterminism explicit and controllable. To this end, we propose a new OS abstraction, the Deterministic Process Group (DPG). A programmer uses this abstraction to define a *deterministic box* inside which all communication happens deterministically. All of the nondeterministic input received by a DPG is interposed upon by the *shim layer*, an interface that can be used by programmers to observe and control external nondeterminism in a flexible way.

A DPG is effectively a high-level deterministic virtual machine. The deterministic guarantees are provided transparently by the OS without intervention from the programmer; thus, DPGs can host arbitrary, unmodified application binaries. At any given time there may be many DPGs running alongside many conventional nondeterministic processes. An alternative design is full-system determinism, in which a hypervisor executes an entire OS deterministically relative to inputs triggered by the hardware. The DPG approach is more flexible because the programmer can select the desired granularity of determinism for each individual application.

1.1 DPG Use Cases

Debugging and Testing Many applications do not continuously interact with the external world, but instead read inputs at deterministic points in their execution. Since DPGs provide internal determinism by default, these applications will execute completely deterministically when run within a DPG. This has obvious benefits for debugging, since execution is directly repeatable. Moreover, removing internal nondeterminism has the potential to reduce the problem of testing multithreaded programs to the problem of testing sequential programs by making execution a function of only the explicit inputs, including external nondeterminism.

Record/Replay Controlling external nondeterminism with the shim layer makes determinism useful even for applications that interact continuously with the external world. As an example, one can run an application inside a DPG and extend the shim layer to log all external nondeterminism. This log can be used later to faithfully replay an application's execution for debugging and other analyses. Most prior work on record and replay of multithreaded applications focuses on how to record internal nondeterminism caused by shared-memory accesses. This leads to either unwieldy logs and high overheads [16, 22] or imprecise replay [1, 31, 36]. The internal determinism offered by DPGs completely subsumes this problem; only external inputs need to be recorded.

Replication for Fault Tolerance DPGs naturally enable replication of multithreaded applications. By running multiple copies of an application inside DPGs on several machines and replicating the inputs, all replicas will behave the same way because there is no internal nondeterminism. This can be implemented by extending the shim layer to ensure that all replicas receive the same input at the same point in their execution. Because DPGs eliminate all forms of internal nondeterminism, there is less to log and replicate. This is a major issue in prior work [5, 38–40] on replication mostly because shared-memory is a very large source of such nondeterminism.

1.2 Outline and Contributions

This paper makes several conceptual and architectural contributions. First, we identify the fundamental distinction between internal and external nondeterminism, and we demonstrate that internal nondeterminism can be eliminated from programs. To do this, we expand on earlier work that removed shared-memory nondeterminism by also removing internal nondeterminism from signals, pipes, the filesystem, and other OS channels.

Second, we propose the Deterministic Process Group abstraction (Section 2), which lets programmers define the boundary between internal and external nondeterminism. As part of this abstraction we introduce the shim layer, whose interface lets programmers observe and control all external nondeterminism.

This paper also presents and evaluates our implementation of these ideas. In Sections 3–4, we describe dOS, a Linux-based implementation of DPGs and the shim layer that enables the deterministic execution of arbitrary, unmodified binaries. Section 5 demonstrates the usefulness of the shim layer by using it to implement deterministic filesystem services, replicated execution of a multithreaded server, and record/replay. Section 6 provides a detailed evaluation of dOS and our shim applications on a variety of workloads. Finally, we end with related work and closing remarks.

2. The Abstraction

Figure 1 illustrates the abstract model of a Deterministic Process Group and Figure 2 illustrates the major components of our system. A DPG consists of a group of threads and processes along with the kernel objects they share. Kernel objects include shared-memory pages, pipes, and sockets. Threads communicate by performing operations on shared kernel objects, for example by reading from a shared page or writing to a shared pipe. A kernel object is *internal* if it can be modified only by threads inside the DPG, and is *external* if it can be modified by threads or devices outside the DPG. We refer to a thread executing inside a DPG as a *deterministic thread*, and we refer to a DPG's set of threads and internal objects collectively as a *deterministic box*.

Figure 1 shows three deterministic threads, $Thread_1$, $Thread_2$, and $Thread_3$, two internal objects, the `memory page` and the `pipe`, and two external objects, the `socket` and the `file`. $Thread_1$ and $Thread_2$ are members of the same process, P_a. The deterministic box is illustrated with a dotted outline. Note that internal objects need not be shared by the entire DPG; in this example, the `memory page` is shared by just two threads.

The final component of a DPG is a user-space service called a *shim program*. A shim program sits on the boundary of a deterministic box, and its job is to interpose on communication that crosses the deterministic boundary. Shim programs are written using a system call interface called the *shim layer*. This interface provides new opportunities for systems programmers that we explore in detail throughout this paper.

2.1 DPGs and Their Guarantees

A new DPG is created with the `sys_makedet` system call, and initially hosts just the calling thread. Each new thread spawned by the initial thread is added to the DPG, and in this way the DPG expands to include all descendant threads and processes. A thread leaves a DPG when it exits. We have not found DPG `join` and `leave` primitives necessary and so have not defined them. Threads

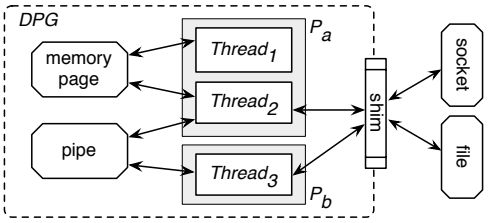


Figure 1. A Deterministic Process Group

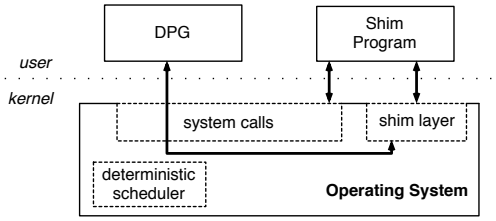


Figure 2. System Overview

```
void shim_attach(tid, SYS|SIG)
void shim_trace(*event)
void shim_resume(tid, result)
void shim_queue_sig(tid, siginfo)
void shim_ctl(tid, ...)
```

(a) Interposing on Nondeterminism

```
void shim_sleep(tid)
void shim_add_barrier(tid, logical_time)
int  shim_gettime(tid)
```

(b) Controlling Logical Time

Figure 3. Shim layer system calls

hosted in a DPG need not share an address space, which means a DPG can host many multithreaded processes.

Deterministic threads invoke system calls and read and write shared-memory just like ordinary threads. However, DPGs distinguish between operations on internal objects, which happen deterministically, and operations on external objects, which happen nondeterministically. Interactions with external objects represent a DPG's *only* source of nondeterminism; essentially, these external interactions represent the inputs a DPG receives from the external world.

Given the same initial state and the same stream of external inputs, a DPG is *guaranteed* to execute the same steps of inter-thread communication and produce the same output. More precisely, as a DPG executes it performs shared-memory `loads` and `stores`, invokes system calls, and handles asynchronous signals; each of these operations introduces nondeterminism *only* when it involves an entity outside the DPG. This is a stronger guarantee than output determinism [1, 23], which guarantees that replaying a program will produce the same output, but not that it will reproduce all inter-thread communication steps that lead to that output.

For example, when operating on a network socket, the `read` system call returns nondeterministic data. Additionally, `read` is a *blocking* call; it does not return until data is available, which means `read` will block for a nondeterministic amount of time. However, when `read` operates on a device that is internal to a DPG, such as an internal pipe, `read` behaves deterministically.

In summary, a DPG experiences nondeterminism only when it: (1) reads data from an external source; (2) blocks to wait for external data; or (3) handles a signal sent from an external source. Our guarantee is that DPGs execute deterministically relative to a stream of such nondeterministic input, and also relative to the initial state of the DPG at the call to `sys_makedet`. Note that this guarantee holds even across different machines.

Logical time Conceptually, a DPG executes as if it was serialized onto a *logical timeline*, where *logical time* is represented by a single global counter. Blocking system calls occupy two points on the logical timeline, one to initiate the call and the other to complete the call. dOS ensures that internal communication is mapped onto the logical timeline in a deterministic way. (Section 3 describes how our implementation groups instructions into atomic epochs in order to extract parallelism.) Note that logical time and physical time are distinct: DPGs guarantee deterministic output, but not deterministic performance. Input from the external world is mapped onto the logical timeline in a way controlled by the shim layer, which is the subject of the next section.

2.2 The Shim Layer

Every DPG is monitored by a user-space service called a shim program, also referred to as a *shim*. Shim programs use the shim layer interface (Figure 3) to observe and control nondeterministic input.

At a high level, there are two kinds of nondeterministic input: the *what* and the *when*. The *what* includes the values of external input, such as data read from the network. Then *when* includes the blocking times of nondeterministic system calls, as well as the delivery times of external signals. Shims can observe and control both kinds of nondeterministic input.

As a motivating example, consider record and replay implemented with a pair of shim programs. The record shim observes execution: for every nondeterministic system call, the shim logs the number of logical time steps the call spent blocked, along with the return value of the call. The replay shim controls execution: it ensures that every nondeterministic system call is scheduled to return at the specific logical time and with the specific value specified in the log.

The following sections first describe how shims observe and control the *what* (Figure 3a), and then how shims observe and control the *when* (Figure 3b), using record and replay as running examples.

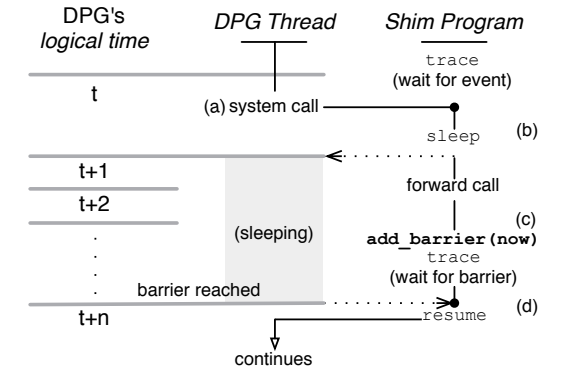


Figure 4. Observing a blocking system call

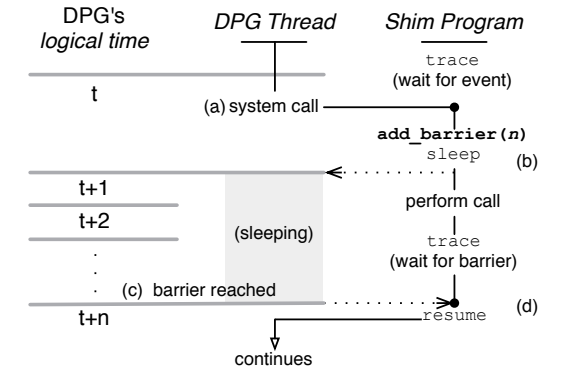


Figure 5. Controlling a blocking system call

2.2.1 Interposing on Nondeterminism

Shims use `shim_trace` to wait for a DPG to encounter nondeterminism. `shim_trace` blocks until either (a) a deterministic thread is about to perform a nondeterministic system call, or (b) an external signal is about to be delivered to a deterministic thread. In both cases, the deterministic thread stalls, execution transfers to the shim program, and `shim_trace` returns. The shim can interpose on this nondeterministic event and then return control back to the deterministic thread by calling `shim_resume`. In this way, execution of a deterministic thread alternates between itself and a shim program, much like execution of an ordinary thread alternates between user-space and kernel-space.

For system calls, `shim_trace` populates the given $event$ structure with the system call number and arguments. The shim should perform the system call on behalf of the deterministic thread and then transfer control back to deterministic thread by calling `shim_resume`, using the $result$ parameter of `shim_resume` to specify the system call's return value. The shim might perform the call by forwarding the call to the OS (*e.g.*, for record) or by ignoring the OS entirely (*e.g.*, for replay).

For external signals, the $event$ structure includes the `siginfo_t` of the pending signal. The shim can queue the signal for delivery by calling `shim_queue_sig`, save the signal internally for later delivery, or discard the signal entirely. In each case, the shim returns control to the deterministic thread by calling `shim_resume` with an empty $result$.

2.2.2 Controlling Logical Time

A shim program monitors the passage of logical time in a DPG by registering logical time barriers using `shim_add_barrier`. A logical time barrier is a timer tied to a specific deterministic thread (through the tid parameter); when the timer goes off, the deterministic thread stalls and the shim is notified through `shim_trace`. The barrier time is specified as an offset relative to the current logical time of the DPG, which can be obtained with `shim_gettime`. Time barriers can be used to control the nondeterministic *when*, as described below.

System Call Blocking Time Figures 4 and 5 illustrate how to observe and control the number of logical time steps that a system call blocks. Both examples follow a similar pattern; the only difference is the way in which `shim_add_barrier` is called.

Figures 4 illustrates observing a blocking system call (*e.g.*, for record). When deterministic thread T performs a system call (a), the call is trapped by the shim, which returns from `shim_trace`. At this point, thread T stalls and the DPG's logical time does *not* advance. The shim can now forward the call to the OS, but before doing so it puts T to sleep by calling `shim_sleep` (b). While T is asleep it is detached from the logical timeline and does not execute; this allows a nondeterministic amount of logical time to pass in the DPG while the system call is being performed. When the system call finally completes, the shim synchronizes with the DPG by registering a time barrier for T to happen at the very next logical time step in the DPG (c). Once that barrier triggers, the shim returns control to T via `shim_resume` (d).

Figure 5 illustrates controlling a blocking system call (*e.g.*, for replay). Again, deterministic thread T performs a system call which is trapped by the shim via `shim_trace` (a). The key difference in this example is that the shim decides, *a priori*, that the system call should complete in exactly n logical time steps. For example, a replay shim would read n from a log. To enforce this, the shim registers a barrier for T that will trigger n steps in the future and then puts T to sleep (b). While T is asleep it does not execute; the rest of the DPG executes normally for exactly n logical time steps, but no further. At this point the barrier triggers: T wakes and notifies the shim (c). Finally, the shim returns from the system call and returns control to thread T (d).

Signal Delivery Time Now suppose a shim wants to deliver a signal to thread T at logical time n. To do this, the shim should simply register a barrier for time n. When that barrier is reached, the shim can queue the

signal for immediate delivery using `shim_queue_sig` and then resume the thread using `shim_resume`.

2.2.3 Shim Use Cases

Shim programs can implement the record/replay and replicated execution services discussed earlier, but we envision many other kinds of shim programs as well. Some shims will be generic, application-independent services written by systems programmers, while others will be written by application programmers and tailored to enhance a specific application. Additionally, a shim program can be used to adjust the boundary of a deterministic box in two ways described below.

Expanding the Set of Deterministic Services An OS that supports DPGs may decide to implement some system calls nondeterministically to reduce kernel complexity, even when deterministic implementations are possible under the right assumptions. For example, in dOS, interaction with local files remains nondeterministic due to variations in disk latency, even though this nondeterminism can be considered internal and thus eliminated under the right assumptions. Section 5.1 explores how a shim can make local file access deterministic.

Further, a shim can virtualize global resources such as process identifiers in a deterministic way, as in [30]. A shim can even convert physical times (*e.g.*, used by `sleep` and `alarm`) into virtual, logical times. This would eliminate nondeterminism introduced by real time, but of course is only meaningful for applications that do not require a precise correspondence with real time.

Customizing the Nondeterministic Interface System calls are a DPG's basic interface to the nondeterministic world. However, it is often beneficial to let applications define the nondeterministic interface at a more abstract level. For example, a server application might want to hide many low-level `read` and `write` system calls behind a single high-level, nondeterministic `getmsg` call. Previous work has argued that this flexibility is valuable for record/replay systems [19], but we consider this flexibility to be even more general; for example, Section 5.3 shows how it is useful for replicated execution.

We enable this flexibility in dOS by defining a new system call, `dpg_callshim`, which makes a direct call from a DPG into its shim. Effectively, `dpg_callshim` allows developers to divide an application into two parts: the deterministic part that runs in a DPG and the nondeterministic part that runs in a shim.

3. Deterministic Execution Algorithm

The first implementation choice we make is which algorithm dOS uses to enforce determinism. Prior work on shared-memory determinism has proposed a family of deterministic execution algorithms, including DMP-O, DMP-B, and DMP-TM [6, 14]. dOS implements the DMP-O ownership-tracking algorithm; we selected it for its relative simplicity of implementation, but any deterministic execution algorithm can support DPGs as long as the shim layer can be implemented on top of it.

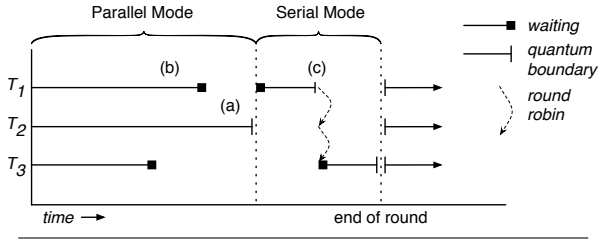


Figure 6. Timeline of a quantum round in DMP-O. T_2 finishes its quantum in parallel mode (a), while T_1 and T_3 have work left for serial mode (b,c).

One constraint imposed by the shim layer is that logical time should be representable with a single global counter. DMP-O, DMP-B, and DMP-TM all satisfy this constraint, but other (as yet uninvented) algorithms may require a more complex notion of logical time, such as a vector clock. We believe the shim layer could be extended to support such algorithms, but the details are left for future work.

In the rest of this section, we first summarize our earlier work on using DMP-O to enforce deterministic execution of multithreaded programs that communicate via shared-memory. Next, we describe how to generalize DMP-O to include communication via channels other than shared-memory, such as pipes and signals.

3.1 Shared-Memory Determinism

Two key observations underlie DMP-O. First, if threads do not touch shared data, *i.e.*, if they do not communicate, their execution will be deterministic no matter how they are scheduled. Second, when threads do communicate, a trivial deterministic schedule is to divide each thread's execution into chunks and then execute all chunks in a deterministic serial order.

Following these observations, execution in DMP-O is divided into chunks called *quanta*. A *round* consists of all threads executing one quantum each. Each round is divided into a *parallel mode* and a *serial mode*. In parallel mode, threads run in parallel but are isolated; they do not communicate. In serial mode, threads run serially but can communicate arbitrarily. A thread ends its parallel mode once it has reached an instruction that might communicate with other threads. Serial mode begins once all threads have completed parallel mode, and ends once all threads have had a chance to run. The parallel and serial modes are thus isolated by global barriers into two-stage rounds, as illustrated in Figure 6.

Notice that the parallel and serial modes are directly inspired from the two key observations stated above. DMP-O is deterministic as long as threads are (1) bro-

ken into quanta at deterministic boundaries, (2) ordered deterministically in serial mode, and (3) correctly isolated in parallel mode. The first two constraints are easily satisfied: we define a quantum to be some deterministic number of *dynamic* instructions, and we order threads in serial mode by sorting them by creation order.

DMP-O achieves isolation in parallel mode by partitioning ownership of shared-memory across threads. Each memory location is in one of two ownership states: *owned-by-T* for some thread T, or *shared*. A location that is *owned-by-T* is private to T; no other thread can access the location during parallel mode. A location that is *shared* is globally read-only; all threads can read the location during parallel mode, but none can modify it. A thread waits for serial mode before performing an operation that does not meet these conditions.

Ownership states evolve during serial mode by following two rules: (1) before thread T writes to a location, it sets ownership of that location to *owned-by-T*; and (2) before T reads a location that is not *owned-by-T*, it sets ownership of that location to *shared*.

Logical time Finally, we say that logical time increments on every *mode transition*, *i.e.*, on every transition from parallel mode to serial mode and back. Note that within a single mode every thread appears to execute atomically. From this property it follows that mode transitions are meaningful increments of logical time.

3.2 Beyond Shared-Memory Communication

Our model of a DPG from Figure 1 is that threads communicate by performing operations on shared kernel objects, which includes more than just shared-memory. To generalize DMP-O to this model we first observe that we can track ownership of shared kernel objects just as for shared-memory locations: if an operation mutates a kernel object it acts as a "write," while if an operation only observes a kernel object it acts as a "read." In fact, our implementation (Section 4.1.1) tracks ownership of shared-memory at the page granularity, effectively treating a memory page as just another kernel object.

To fully generalize DMP-O we need two additional changes: the first deals with blocking operations, and the second deals with asynchronously delivered signals.

Blocking Operations When a system call blocks, the calling thread ends its current mode (either parallel mode or serial mode) and is not scheduled to run again until it unblocks. While a thread is blocked, the rest of the DPG continues to execute. A thread can only unblock during a mode transition; this ensures that threads unblock at discrete points on the logical timeline.

Signal Delivery Incoming signals are queued during the current mode then delivered immediately on the next mode transition. Queued signals are partitioned into *internal* and *external* signals, depending on whether they were sent from a thread inside or outside the DPG, respectively. If there are N threads in a DPG then each deterministic thread has N logical queues: one queue for external signals and one queue for signals sent from each of the $N-1$ other deterministic threads.

On a mode transition, internal signals are delivered first and external signals are delivered last. The internal signal queues are emptied in a deterministic order, *e.g.*, by using the ID of the sending thread as a sort key.

This strategy ensures, first, that internal signals are delivered deterministically, and second, that external signals are delivered at meaningful logical times. Note that when a thread sends a signal to itself (as with `SIGSEGV`) the signal is synchronous; such signals are always delivered instantly. dOS implements the N-queue model described here using a single sorted list. Additionally, dOS always delivers external `SIGKILL` signals immediately (rather than forwarding them to the shim) so that a DPG can be killed even when its shim program misbehaves.

4. Linux-Based Implementation

We now describe how we implemented dOS, which is a variant of Linux that implements the DPG abstraction. dOS makes two major changes to Linux: first, it implements the shim layer; and second, it implements DMP-O, which includes an object ownership-tracking mechanism and a deterministic scheduler that constrains the execution of each DPG to a deterministic logical timeline. dOS exports a traditional system call interface to DPGs along with the `sys_makedet` and `dpg_callshim` system calls.

Our implementation of DMP-O was the most challenging and invasive change. Overall, we added roughly 5800 lines of new code to the Linux 2.6.24-7/x86-64 kernel and changed roughly 2500 lines of existing code in 53 files. Below we summarize the low-level implementation details of dOS and discuss engineering challenges (Sections 4.1–4.4). We end with a summary of the strengths and limitations of our implementation (Section 4.5).

4.1 Ownership Tracking

4.1.1 Shared-Memory Pages

dOS tracks ownership of shared-memory at the page granularity by using hardware page-protection to verify that a deterministic thread does not access a page without appropriate ownership.

Conceptually, dOS maintains a *shadow page table* for each thread. A thread's shadow table mirrors its real page table exactly, except that shadow permission bits are modified to reflect the current distribution of page ownership. dOS exposes only the shadow page tables to hardware: on a context switch to thread T, dOS installs T's shadow table onto the CPU even if the previously scheduled thread shared an address space with T.

Page ownership is encoded into shadow page table permissions so that ownership violations such as a `store` to a *shared* page will trigger a page fault. dOS intercepts this page fault, notices it is due to an ownership violation, and stalls the faulting thread until it is scheduled to run in serial mode. dOS then assigns ownership of the page to the faulting thread and continues its execution.

Every conventional process has one real page table representing its *address space*. All address space modifications are expressed in terms of the real page table and then transparently applied to the shadows. To limit memory overheads, dOS maintains just N shadow tables per address space, where N is the number of CPUs, and then assigns threads to shadow tables, effectively bucketing the threads in a given process into N ownership groups. This requires a slight tweak to the DMP-O scheduler: during parallel mode, all threads that share a shadow table must be serialized in a deterministic order (*e.g.*, scheduled serially in thread creation order). We bucket threads using a simple greedy algorithm.

This strategy is not limited to shared-memory within a single process. dOS supports shared-memory across processes by tracking ownership of physical pages; we use Linux's `rmap` facility to enumerate all user-space addresses that map a given physical page.

Finally, there are two corner cases worth mentioning. First, dOS disables address space randomization for DPGs so that every DPG has a deterministic address space layout. Note that we can enable address space randomization in DPGs if we expose the seed as external nondeterminism. Second, page swapping can introduce nondeterministic changes to page tables. To preserve determinism, when a page is swapped out, dOS preserves the page's ownership state using extra bits in the shadow page tables. When a page fault triggers a swap-in, dOS stalls the thread until the page is read from disk, and then restores the saved ownership state of the page.

4.1.2 Other Kernel Objects

Other kernel objects, such as pipes and sockets, are operated on by system calls. dOS instruments the kernel so that a system call never operates on a kernel object unless the calling thread has the appropriate level of ownership.

Adding this instrumentation presents two engineering challenges. First, where should the instrumentation be placed? It is tempting to lazily acquire ownership of an object just before a system call actually uses the object, but doing this requires reengineering kernel locking protocols. To see why, note that acquiring ownership may require sleeping the calling thread to wait for serial mode. However, a system call may not decide to use an object until inside an atomic region, *e.g.*, while holding a spin lock, and it is not safe to sleep in such regions.

dOS avoids this difficulty by conservatively acquiring ownership of all objects a system call may use before executing the call. This requires adding instrumentation in just two places: at the system call entry point, and in the code that wakes up a thread. dOS instruments thread wakeup to reacquire any privileges lost while the system call was asleep, *e.g.*, while waiting for input.

Behavior	(Total Syscalls) Examples
use *pages*	(14) mprotect, read
use *address space*	(6) mprotect, mmap, brk
use *inode*	(32) read, write, lseek, close
use *fd table*	(9) open, dup, close
use *fs path*	(22) open, chdir, chroot, access
read *untracked*	(54) getpid, gettimeofday
modify *untracked*	(172) kill,setrlimit,sigaction

Table 1. System call behaviors

The second challenge is that Linux is a large, complex system with over 250 system calls and many unique types of kernel objects. To simplify our implementation, we track a few kinds of kernel objects precisely and then conservatively merge all other kinds of objects into an *untracked objects* group. For all but the untracked objects, dOS tracks ownership using a hash table that maps an object to its current owner. Freshly allocated objects are initially *owned-by* the allocating thread. Ownership of the untracked objects is implicit: during parallel mode they are *shared*; and during serial mode they are *owned-by* the thread currently running. Thus, read-only operations on untracked objects can execute in parallel mode, while all other operations on untracked objects must wait for serial mode. This strategy is summarized in Table 1.

An `inode` is Linux's internal name for files, sockets, pipes, and anything else that can be referenced by a file descriptor. System calls like `read` that operate on file descriptors can modify the contents of memory pages, map new pages into the address space, or even modify the `inode` itself. These system calls must acquire ownership of all of these objects before proceeding.

4.2 Scheduling

The dOS scheduler is implemented as a filter in front of the default Linux scheduler—it does not push a deterministic thread into the Linux scheduler until the thread has been scheduled to run by its DPG. This filter implements the DMP-O scheduling algorithm.

Thread Creation The `fork` and `clone` system calls always execute in serial mode. This ensures that deterministic threads are spawned in a global serial order. The newly spawned thread will be scheduled to run during the next parallel mode.

Logical Time Barriers The dOS scheduler checks for pending time barriers on each mode transition. To prevent deadlock, dOS instantly fast-forwards logical time to the next pending time barrier whenever all threads in a DPG are simultaneously asleep.

Quantum Formation Recall that parallel mode ends when all threads have either reached a quantum boundary or stalled to acquire ownership, and serial mode ends once all threads have reached a quantum boundary, where quantum boundaries must occur at deterministic points in a thread's execution. A possible implementation is to mark quantum boundaries with system calls, but this does not guarantee forward progress because a thread may loop forever without making any system calls. Additionally it does not guarantee *balance*; imbalance leads to excessive waiting at the end of parallel mode, which leads to poor performance [6].

To guarantee forward progress, dOS defines a *quantum budget*, which is the maximum amount of work a thread can perform in a quantum. dOS estimates work by counting instructions. The quantum budget is simply a deterministic number of instructions, typically in the range of tens to hundreds of thousands of instructions.

dOS counts instructions using the hardware "instructions retired" counter that is available on all modern x86 CPUs. dOS configures this counter to trigger an overflow interrupt after the quantum budget expires. There are well-documented caveats about using this counter [15, 43]. Specifically, the counter suffers from nondeterminism that can be engineered around. We follow the solution outlined by [15]: to overcome imprecise interrupt delivery, dOS must single-step the DPG (via the x86 trap flag) for up to about 200 instructions per quantum, which can introduce large overheads. To avoid those overheads, as an optimization, dOS deterministically ends a quantum when returning from a system call if the remaining quantum budget is low, but not yet exhausted.

4.3 Additional Optimizations

As demonstrated in [6], DMP-O performs best when parallel mode is balanced and when serial mode is empty. dOS implements a few optimizations to bias execution towards these conditions. dOS automatically adjusts a DPG's quantum budget: when dOS detects significant parallel mode imbalance, the budget is decreased to reduce imbalance, and when dOS detects well-balanced parallel modes, the budget is increased to reduce quantum barrier overheads. To limit the time spent executing in serial mode, dOS ends a quantum after a few (heuristically determined) ownership transfers. All of these optimizations preserve determinism, since the parameters used evolve deterministically.

4.4 Shim Programs

In concrete terms, a shim program is composed of a collection of threads called *shim threads*. Shim threads begin life as ordinary user-space threads, *e.g.*, after being spawned by `fork` or `clone`. An ordinary thread becomes a shim thread by calling `shim_attach` to attach to some deterministic thread T. Once attached to T, the shim thread is the distinguished thread that will intercept all of T's nondeterminism through `shim_trace`. If the shim thread crashes, T will stall on external operations until attached to by another shim thread.

A thread can act as a shim thread for more than one deterministic thread. Additionally, to simplify the implementation of shims, a shim thread can elect to receive only the nondeterministic system calls or only the external signals for a given deterministic thread (by setting the second parameter of `shim_attach`). Our usual strategy is to spawn one shim process for every DPG. Within this process we spawn one shim thread to intercept signals for the entire DPG, and for every deterministic thread in the DPG we spawn one shim thread to interpose on the system calls performed by the corresponding DPG thread.

Intercepting System Calls When a shim program intercepts a system call it has two options: (1) it can emulate the system call completely; or (2) it can simply instrument the system call's entry and exit, allowing the deterministic thread to actually execute the body of the system call. These options resemble those allowed by `ptrace`.

The option to simply instrument a system call is selected by passing a special result to `shim_resume`. This option gives a shim limited control over how the system call executes in logical time. For example, if a shim simply instruments `read` instead of emulating it, the shim cannot observe or control when the kernel writes to the given user-space buffer (the writes will happen nondeterministically, in an unrecordable way). We provide instrumentation as a convenience for cases where full emulation is not necessary. During system call emulation, a shim can use `shim_ctl` to perform side effects in a DPG, such as writing to or reading from a user-space buffer.

RDTSC dOS allows shim programs to interpose on the nondeterministic `RDTSC` instruction. Our implementation uses the time stamp disable flag of the x86 `cr4` register to fault on user-mode accesses to `RDTSC`; these events are exposed to the shim via `shim_trace`.

4.5 Discussion

Guarantees Provided by dOS dOS guarantees that communication via the following kernel objects is deterministic as long as the objects are completely internal to a given DPG: shared-memory pages, including across multiple processes; pipes allocated with `pipe`; and `futexes` (used to implement pthreads synchronization). Additionally, dOS guarantees that file descriptors and memory pages are allocated in a deterministic order; that the address space evolves deterministically (as via `mmap`); that internal signals are delivered deterministically; and that `wait` is deterministically notified when threads in the same DPG exit.

Note that some system calls are deterministic except in error cases. For example, `mmap` allocates pages deter-

ministically within an address space, but will fail nondeterministically if there is not enough physical memory available to service the request.

Guarantees Not Provided by dOS Our deterministic guarantees may not translate across different versions of program binaries no matter how slightly different (*e.g.*, after a patch). Also, although our guarantees hold across different host machines, an application can read host configuration as part of its inputs, for example to dynamically adjust its resource usage; these inputs must be duplicated exactly to guarantee determinism.

Additionally, dOS does not guarantee deterministic access to shared-memory pages that can be modified by threads or devices outside the DPG. Ideally we might interpose on this external communication using the shim, but this would require adding excessive restrictions to non-DPG processes. For example, page ownership might transition between "exclusive to a DPG" and "exclusive to the external world," but this would require stalling external threads as they wait to reacquire page ownership. Relatedly, DPGs may encounter nondeterminism when memory is modified through backdoors in `/proc`.

Retrospective Implementing DPGs in a monolithic kernel such as Linux raises many thorny issues. The example of `mmap` is instructive: reasoning about the cases in which `mmap` is nondeterministic requires finding and reasoning about many code paths in a monolithic kernel.

More generally, providing determinism requires tracking and mediating accesses to shared OS objects. However, many Linux kernel objects have aliased names, are named in multiple namespaces, and are accessible through multiple interfaces. For example, process IDs are exposed through system calls, the `/proc` filesystem interface, and in some cases, thread-local storage variables in the address space of a multithreaded process. If we consider PIDs to be a source of internal nondeterminism, dOS must correctly track and reconcile PIDs through all of these channels, for instance, by virtualizing PID numbers before they are exposed to a program so that PID assignment is deterministic and consistent across processes within a DPG. Even if we consider PIDs a source external nondeterminism (the choice made by dOS), for record/replay to work correctly a shim program must interpose on all of these different channels for accessing PIDs, so that PIDs can be recorded and during replay the same PIDs can be reassigned.

An OS kernel implemented "from scratch" to support DPGs would benefit from design principles advocated by exokernels and microkernels. A minimal kernel interface combined with a libOS would push many of the aliased interfaces and complex code paths out of the kernel and inside the user-space deterministic box, making it easier to reason about determinism at the system call layer. The protection domains of a microkernel could further simplify many of these issues, since reasoning about nondeterminism would largely reduce to detecting messages that cross the boundary of a deterministic box. In the `mmap` example, this might be a message to the page-allocation server.

5. Shim Applications

To demonstrate the usefulness of the shim layer, we have implemented three shims: deterministic filesystem services; record/replay by logging just external input; and replicated execution of a multithreaded server. The deterministic filesystem service and record/replay shims can be used with unmodified application binaries, while the replicated execution shim is application specific. We note that the shim layer allowed us to quickly prototype the shims described in this section.

5.1 Deterministic Filesystem Services

FSSHIM provides applications with a *deterministic file hierarchy*. All reads and writes to files within this hierarchy are deterministic; accesses to files outside of this hierarchy are considered sources of external nondeterminism, as before. There are two sources of nondeterminism FSSHIM must eliminate: the latency of each operation, and the number of bytes operated on by the `read` and `write` system calls. FSSHIM eliminates the first by deciding, *a priori*, that each operation will block for a fixed and deterministic amount of logical time. For the second, FSSHIM guarantees that all `reads` and `writes` operate on a deterministic number of bytes by always performing the maximum amount of work requested (up to an end-of-file, for `reads`). FSSHIM can make these guarantees because it performs the `read` and `write` calls on behalf of the DPG, using the pattern illustrated in Figure 5.

The deterministic blocking time selected by FSSHIM can affect performance. For example, if FSSHIM selects a logical blocking time that is too low, the DPG will stall waiting for disk operations to complete. On the other hand, if FSSHIM selects a time that is too high, the calling thread will execute artificially slowly. The logical blocking times we chose for FSSHIM are equivalent to a delay of about 5 million instructions; we did not experiment heavily with this number.

A file can exist in the deterministic file hierarchy only if it can be considered *internal* to the DPG, which is true when: (1) the initial contents of the file are deterministic; (2) the file is not written by any threads outside the DPG; and (3) operations on that file complete in a finite time. In practice, the third assumption implies fail-stop. FSSHIM relies on the user to explicitly indicate the parts of the filesystem for which these assumptions are valid. This typically includes the directories containing program inputs, as well as directories shared system-wide that are rarely updated, such as `/usr` and `/etc`.

5.2 Record/Replay

RECSHIM records all external nondeterminism introduced through the system call interface and signals, enabling deterministic program replay. RECSHIM needs to record only the external nondeterminism because DPGs eliminate all forms of internal nondeterminism. Further, RECSHIM can be combined with FSSHIM, reducing what needs to be logged since accesses to files within the deterministic file hierarchy would be deterministic.

RECSHIM utilizes the shim layer to interpose on system calls and to intercept external signals. System calls that touch user-space memory are executed by RECSHIM on behalf of the DPG. RECSHIM produces a log containing the logical time the event occurred and any other event-specific information needed during replay. For system calls, this includes the return value and logical blocking time, as well as any side-effects of the system call, such as the contents of a buffer after performing a socket read. For signals, a copy of the `siginfo` is saved. Logs are compressed on-the-fly with zlib.

We have implemented a proof-of-concept replay shim to verify that the shim layer offers all the hooks necessary to implement a replay component. The major challenges in faithfully replaying system call traces are orthogonal to the main body of our work and have been explored by prior work [19, 36, 37].

5.3 Replicated Execution

REPLICASHIM supports replication of a multithreaded webserver running inside a DPG by guaranteeing that the order of messages and their logical arrival time is kept consistent across all replicas. Given the same inputs and the same logical arrival times, the DPG abstraction guarantees that all replicas will evolve deterministically.

Our target application is `nullhttpd` [12], a small, simple, multithreaded webserver that uses a thread-per-request model. Our design splits the functionality of the basic server into three separate process types: a single arbiter process, and a set of replicas, each composed of a shim process along with a DPG that hosts `nullhttpd`.

The arbiter process operates nondeterministically, outside of any DPG, and accepts incoming HTTP requests from the network. The arbiter broadcasts requests to the replicated shims, which queue the requests locally. We modified `nullhttpd` to read new requests by making a direct call to its shim via `dpg_callshim`, rather than reading from the network. This shows a case where the programmer defines the interface via which nondeterministic inputs are received.

When the arbiter broadcasts a request, it must ensure that all replicas see that request at the same logical time. It does this by performing a two-phase commit to determine a logical time that no replica has advanced beyond. The protocol works as follows. When the arbiter receives a new HTTP request from the network, it asks all replicated shims to set a barrier and report their current logical time. The arbiter uses the maximum value reported by any replica as the logical arrival time of the new request. The arbiter then broadcasts the new request and asks each replica's shim to set a second barrier for this arrival time; once this barrier is reached at a replica, the replica's shim makes the new request available to `nullhttpd` and the replicas continue to evolve deterministically.

6. Evaluation

The goal of our evaluation is to understand the performance of DPGs in comparison to ordinary nondeterministic execution (`Nondet`). We include evaluations of the three shim programs we built, namely FSSHIM, RECSHIM, and REPLICASHIM.

Correctness We tested our dOS implementation by running the `racey` [20, 45] deterministic stress test 500 times and verifying that `racey` always produces the same output. In addition to the basic `racey` program, we tested `racey` variants that exercise the various components of our implementation, such as communication via pipes, signals, and multiprocess shared-memory.

Workloads We evaluated the following parallel workloads: the `PARSEC` [8] and `SPLASH2` [44] benchmark suites; `pbzip2` [18] to compress a Linux ISO image; and `make -j` to perform a parallel build of the Linux kernel. The `PARSEC` and `SPLASH2` are workloads optimized for parallelism; we scaled their inputs to run for about a minute with a single nondeterministic thread. We present a representative subset of the `PARSEC` and `SPLASH2` benchmarks that was selected to showcase both the best-case and worst-case performance of dOS.

We also evaluated three reactive applications: the Apache and `nullhttpd` webservers and the Chromium web browser. Apache and Chromium are especially interesting because they use multiple processes with multiple threads per process. We evaluated the webservers using httperf [26] to simulate a constant stream of requests for static pages. We evaluated Chromium with two experiments: first, we measured the load time of `nytimes.com` (without any local caching); and second, we used Chromium's debugging facilities to execute a scripted user session that opened 5 tabs and navigated to 12 URLs in rapid succession. All Chromium experiments used the process-per-tab model [34].

We ran our experiments on 8-core 2.8GHz Intel Xeon E5462 machines with 10GB of RAM using `rundet`, a small utility that constructs a single DPG and then executes an unmodified application binary inside that DPG. We used a relatively aggressive machine configuration to adequately explore the scalability of our parallel workloads. All results shown are the average over ten executions, with the highest and lowest values removed.

Benchmark	Config: Num Proc	Config: Threads per Proc	Throughput: Nondet	Throughput: DPG only	Throughput: DPG + FSSHIM
apache 10KB	16	1	10.1K	3.6K	1.7K req/s
apache 10KB	4	4	10.1K	6.6K	2.2K req/s
apache 10KB	1	16	10.2K	7.4K	2.4K req/s
apache 100KB	4	8	1.1K	1.1K	0.9K reqs/s
nullhttpd 10KB	1	16	1.0K	1.0K	1.0K req/s
chromium	nytimes		1.8 s	2.4 s	3.8 s
chromium	scripted		22 s	37 s	40 s

Table 2. Reactive Workload Evaluation

Benchmark	Overheads (relative to Nondet): DPG Only 2-th	DPG Only 4-th	DPG Only 8-th	DPG + FSSHIM 2-th	DPG + FSSHIM 4-th	DPG + FSSHIM 8-th	Speedup (8-th over 2-th): Nondet	Speedup (8-th over 2-th): FSSHIM
blackscholes	1.2	1.2	1.3	1.2	1.3	1.3	3.4	3.2
dedup	2.3	3.6	4.0	4.0	5.8	6.4	1.6	1.0
fmm	2.6	6.1	10.1	2.6	6.0	10.1	2.4	0.6
lu	2.0	2.3	2.3	2.0	2.3	2.3	2.1	1.7
pbzip2	2.0	2.7	3.0	2.1	2.8	3.4	2.6	1.6
make	2.3	4.1	5.9	3.2	5.7	8.2	2.8	1.1

Table 3. Parallel Workload Evaluation

6.1 DPG Overheads

We start with two questions: what are the overheads of DPGs for typical workloads, relative to nondeterministic execution, and how much overhead is added by FSSHIM? To answer these, we ran our workloads in DPGs with no shim attached and in DPGs with FSSHIM attached.

Table 2 summarizes this evaluation for reactive workloads. The first few rows evaluate Apache for workloads of 10KB and 100KB static pages. For the 100KB workload, both the `Nondet` and the DPG-only case are able to saturate the gigabit network of the Apache server, in spite of the extra overhead of using the DPG. FSSHIM adds some additional overhead, enough to shift the system bottleneck to the CPU.

For the 10KB workload, the `Nondet` case is still able to saturate the network link. However, this workload involves a significantly higher rate of system calls and other nondeterministic events; each system call incurs a context switch from the DPG to its shim. As a result, both the DPG-only and the FSSHIM cases experience serialization and overhead that slows the request rate between 1.4x and 5.9x.

Throughput generally decreases as the number of processes (Column 2) increases. We suspect this is because interprocess communication is more costly when executing in a DPG. Note that scaling can be achieved by running multiple smaller instances of Apache in separate DPGs. Overall, we consider these throughputs reasonable for all but the most high-traffic web sites.

The last two rows show the execution time of Chromium. For the scripted session, latency increases by 1.7× for DPGs alone and by 1.8× for DPGs with FSSHIM. Latency increases from 1.8 seconds to just 2.4 seconds when loading `nytimes.com`. We also performed this test for a Google search results page (not shown). All execution times in the Google search results test were less than a second, and informally, the differences "felt" negligible when we interacted with the browser.

Benchmark	Config: Num Proc	Config: Num Thread	Exec Breakdown: % Serial Mode	Exec Breakdown: % Single Stepping	Serialization Reasons: % Pgfault	Serialization Reasons: % Syscall
apache 10KB	16	1	72%	< 1%	< 1%	99%
apache 10KB	4	4	80%	< 1%	< 1%	99%
apache 10KB	1	16	82%	< 1%	< 1%	99%
apache 100KB	4	8	26%	< 1%	2%	98%
nullhttpd 10KB	1	16	11%	0%	2%	98%
chromium	nytimes		58%	13%	61%	39%
chromium	scripted		25%	13%	72%	28%
blackscholes	1	8	3%	27%	99%	1%
dedup	1	8	54%	12%	77%	23%
fmm	1	8	90%	18%	100%	0%
lu	1	8	45%	35%	95%	5%
pbzip2	1	8	35%	39%	100%	0%
make	8	1	79%	3%	0%	100%

Table 4. DPG Execution Characterization

Table 3 shows execution overheads for our parallel workloads with 2, 4, and 8 threads. Overheads are generally below 3×, often lower than 2.5×. Columns 5-7 show the added cost of FSSHIM, which is typically small, since most of the applications do not perform a significant number of system calls (except `dedup` and `make`).

DPG scalability is closer to `Nondet` scalability when the overheads do not grow much with the number of threads. Scalability suffers for workloads like `fmm` that share frequently at finer than page-level granularity, but `blackscholes`, which does not have fine-grained sharing, has DPG scalability very close to `Nondet`.

Characterization Table 4 characterizes execution with DPGs with FSSHIM attached. Column 4 shows the fraction of time execution was serialized (*i.e.*, in serial mode). As expected, for the parallel workloads, serialization is highly correlated with overheads and scalability. `blackscholes` and `fmm` are good comparison points; `blackscholes` is 3% serialized and scales nearly ideally with DPGs, while `fmm` is 90% serialized and has poor scalability. For the reactive workloads, the relationship between serialization and performance is less clear, as shim context switch overhead and quantum imbalance are also important factors. The rightmost set of columns show the reason for serialization, broken down into ownership page faults (Column 6) and system calls (Column 7). In reactive workloads, most serialization happens due to system calls, which is expected because reactive workloads perform frequent I/O. Conversely, for parallel workloads (except `make`), most serialization is due to ownership page faults. Also, the fact that dOS uses page-level ownership tracking can lead to unnecessary serialization due to false-sharing.

Even though serialization is very low in blackscholes, the overheads are still on the order of 30%, largely because of single-stepping. Column 5 shows the fraction of execution during which at least one thread is single-

Benchmark	Overheads w/ FSSHIM	Overheads w/o FSSHIM	Log Sizes (per day) w/ FSSHIM	Log Sizes (per day) w/o FSSHIM	Log Sizes (per day) SMP-ReVirt (from [16])
fmm	6.0	6.0	1.1 MB	2.0 MB	83.6 GB
lu	2.4	2.4	11.0 MB	13.0 MB	11.7 GB
ocean	3.0	3.0	1.5 MB	3.6 MB	28.1 GB
radix	4.5	4.5	0.8 MB	2.1 MB	88.7 GB
water	4.8	4.8	5.3 MB	83.2 MB	58.5 GB
pbzip2	2.9	4.0	5.7 MB	295.7 GB	—

Table 5. RECSHIM for Parallel Workloads (4 threads)

Benchmark	Config Num Proc	Config Threads per Proc	Throughput w/ FSSHIM	Throughput w/o FSSHIM	Log Sizes w/ FSSHIM
apache 10KB	16	1	1.7K req/s	1.6K req/s	48.6 B/req
apache 10KB	4	4	2.3K req/s	2.1K req/s	51.3 B/req
apache 10KB	1	16	2.2K req/s	2.2K req/s	50.4 B/req
chromium	nytimes		4.2 s	3.9 s	600 KB
chromium	scripted		40 s	43 s	3.3 MB

Table 6. RECSHIM for Reactive Workloads

Config	Throughput 1 replica	Throughput 2 replicas
`Nondet`	386 req/s	373 req/s
REPLICASHIM	369 req/s	372 req/s

Table 7. Replicated Execution Overheads

stepping; this varies from 0% to 39%. One interesting trend is that reactive applications single-step less often; these applications perform system calls frequently, which triggers an optimization to end quanta early (Section 4.2). Note that single-stepping does not necessarily correlate with performance because serialization and quantum imbalance dominate. In addition to data shown here, we measured the increase in frequency of total page fault events due to ownership changes. While the frequency is often higher, it was not directly correlated with performance. Serialization, quantum imbalance, and single-stepping are the dominant factors.

In summary, DPG overheads are reasonable for several applications, including some parallel applications and most reactive applications. Broadly, overhead tends to increase with sharing, especially as the number of threads grows. We did not attempt to optimize applications for more "determinism friendly" sharing, which could improve performance.

Microbenchmark To more closely understand the overhead of intercepting system calls with a shim, we wrote a simple benchmark that does nothing but call `getpid` in a loop. We ran this benchmark both in a DPG without a shim, and in a DPG with a "null-shim." The null-shim configuration ran 5× slower, suggesting that dOS imposes an overhead of 5× on system call entry.

6.2 RECSHIM: Execution Recorder Shim

We next evaluated the overhead of using RECSHIM, and its resulting log sizes. Table 5 characterizes RECSHIM for parallel workloads. Columns 2-3 show the overheads for RECSHIM with and without a deterministic file hierarchy, respectively. These overheads are essentially identical to execution without RECSHIM (Table 3). Columns 4-5 show log sizes for a full day of execution. RECSHIM's log sizes are very small because DPGs eliminate internal nondeterminism; the remaining nondeterminism is due to a few system calls such as `gettimeofday`.

Not making filesystem accesses deterministic (Column 5) increases the sources of nondeterminism, leading to larger logs. This is especially true for `pbzip2`, which must log the entire ISO image. These log sizes, however, are still orders of magnitude lower than the sizes reported by SMP-ReVirt [16] (Column 6). This is because SMP-ReVirt needs to record internal nondeterminism (again, especially shared-memory), which can be massive. Since SMP-ReVirt is a hypervisor, it logs nondeterminism internal to the OS, adding overhead for `radix` that a process-level implementation of SMP-ReVirt might be able to avoid. This is also an indication that determinism enforcement at the hypervisor level is likely to have a higher performance cost than when enforced at the process level.

Table 6 shows overheads and log sizes for RECSHIM when running reactive applications. Columns 4-5 show the throughput while recording, both with and without FSSHIM enabled. With FSSHIM enabled, RECSHIM did not reduce the throughput of the webservers from the results shown in Table 2. However, disabling FSSHIM resulted in a small performance decrease. The decreased performance is due to the overhead of logging additional input. The overheads for Chromium are about the same as those seen in Table 2. Column 6 shows log sizes normalized to the number of requests for the Apache runs, as well as total log sizes for Chromium sessions.

6.3 REPLICASHIM: Replicated Execution Shim

We end our evaluation by investigating whether we can we quickly build on DPGs to enable replication of an existing multithreaded application. To answer this, we built and tested REPLICASHIM, which replicates our modified `nullhttpd`. For a performance comparison, we also ran replicas outside DPGs but still using the same arbiter and replication protocol (`Nondet`). This configuration does not provide any deterministic guarantees. Table 7 shows the throughput for 1 and 2 replicas with 16 threads per replica; each replica ran on a separate machine while the arbiter ran on a third machine. In both cases, the throughput is essentially matched. Note we did not spend much time optimizing the arbiter or its simplistic protocol, as REPLICASHIM is only as a proof-of-concept; the arbiter is the major bottleneck in these experiments.

6.4 Summary

Our evaluation illuminated the impact of determinism on application performance and scalability. Workload is fundamental factor: applications with frequent inter-

thread or inter-process sharing will encounter more overhead and worse scalability when executed deterministically, since this communication must be tracked and controlled. Implementation choices also have a large impact. We suspect that much of the overhead in dOS is not fundamental and might be mitigated by using sharing-aware memory allocation, by fine-tuning integration with the Linux scheduler, or by using potential upcoming hardware support for transactional memory [2].

The choice of deterministic execution algorithm is another factor. Algorithms like DMP-O that provide a strict memory model or make heavy use of barriers will likely perform worse than those that that loosen the memory model or rely on alternative mechanisms such as speculation. dOS could have implemented the DMP-TM and DMP-B algorithms we developed in earlier work [6, 14]. Both algorithms have better demonstrated scalability than DMP-O and can both be implemented at the kernel level, but both algorithms are more complex. The key idea of DMP-B is to relax the memory model by using a store buffer, which allows concurrent writes in the same quantum round and therefore improves scalability. The key idea of DMP-TM is to use transactional memory to speculate that each quantum round is conflict-free and thus can be executed in completely parallel.

7. Related Work

Deterministic Execution There are a few recent proposals for removing internal nondeterminism in multithreaded execution. DMP [14] is a hardware proposal that includes two approaches for deterministic execution: DMP-O uses ownership tracking at a cache-line granularity; DMP-TM uses transactional memory [33] to further reduce the cost of determinism by speculating that there is no communication between threads. Kendo [29] proposes a software-only library that provides a set of deterministic synchronization operations that offer some deterministic guarantees for race-free programs. CoreDet [6] proposed DMP-B and used compiler and runtime system to provide determinism for arbitrary C/C++ programs. Grace [7] uses speculative execution to provide determinism for fork-join parallel programs. These proposals all describe algorithms for *execution-level determinism*, as used by DPGs. Unlike these prior proposals, however, DPGs support determinism beyond shared-memory in arbitrary binary programs and also provide a way to precisely control external nondeterminism.

Another approach is *language-level determinism*, which uses a parallel language that is deterministic by construction, such as StreamIt [41], SHIM [17], NESL [10], Jade [35], or DPJ [11]. The prime trade-off between execution-level and language-level determinism is one of generality and controllability. In language-level determinism, the programmer must use specific language constructs but gets explicit control of which deterministic executions are possible; in execution-level determinism the programmer can use any language (*i.e.*, determinism is fully transparent) but cannot control which deterministic executions will happen, making behavior less predictable at program construction time. While deterministic languages are a promising long-term solution, the majority of today's programs are written in mainstream languages such as C++ or Java, and this will likely remain the case for the foreseeable future. Additionally, parallel languages are often domain-specific and not well suited to general purpose, reactive applications; in contrast, we have used dOS to demonstrate how reactive applications can benefit from execution-level determinism.

Determinator [3] proposes to enforce determinism using a custom microkernel. Like dOS, Determinator supports multiple processes and uses page protection to enforce determinism of shared-memory accesses. Determinator supports both standard pthreads programs, via an implementation of DMP-B, as well as programs written using specialized parallel programming constructs that are designed to be deterministic. Unlike dOS, however, Determinator does not explore the separation between internal and external nondeterminism, and further, Determinator has no equivalent of the DPG shim layer interface for precisely controlling external nondeterminism.

Record/Replay Record and replay is a natural way to cope with internal nondeterminism during debugging. There are many proposals for software-based implementations of record and replay. Some record all shared accesses that lead to communication [22]; others assume uniprocessor execution and record only scheduling decisions [13]; others record only synchronization operations [36]. The high overheads of logging shared-memory communication motivated several proposals for hardware-supported recording [24, 45, 46], including some recent OS work on virtualization of hardware mechanisms for recording [25].

More recent work [1, 31, 47] relaxes the guarantees of replay by recording just a subset of the information required for faithful deterministic replay. The result is a smaller log at the cost of requiring a potentially impractical search of the execution space during replay. ESD [48] uses symbolic execution to reconstruct thread schedules given only a core dump, without requiring any execution logs to begin with. Unfortunately, ESD suffers from the incompleteness problems faced by symbolic execution, and thus cannot guarantee that a suitable execution will be found during replay.

Two recent and notable record/replay systems are SMP-ReVirt [16] and Scribe [21]. Both systems use page ownership similarly to dOS but record ownership transitions rather than imposing a single deterministic order, as in dOS. SMP-ReVirt is a hypervisor, and so it sup-

ports full-system replay only, while Scribe is a kernel extension, allowing it to support replay of process groups much like dOS. Additionally, Scribe and dOS use similar strategies to track ownership changes of kernel objects.

In contrast to all record/replay systems, the determinism guaranteed by DPGs enables precise replay without needing to record any internal nondeterminism.

Replicated Execution Most prior work in multithreaded replicas has taken the approach of recording and replicating internal nondeterminism. Examples include systems that assume a uniprocessor [28, 40]; that assume race-freedom [4, 5]; and that conservatively replicate all potential shared-memory nondeterminism [39]. Recently, Replicant [32] proposed a limited form of deterministic execution specifically for the purpose of deterministic replication, but this approach requires programmer annotations. Most recently, Respec [23] executes replicas independently while periodically verifying consistency; when consistency is violated, replicas are rolled back to a consistent state and execution proceeds more conservatively. Respec does not support replication across more than one machine, limiting its usefulness. In contrast to prior systems, the determinism offered by DPGs naturally enables replication.

There are some parallels between how dOS provides deterministic execution within a process group and how toolkits like Isis [9] and Horus [42] provide virtually synchronous execution to a distributed process group. Isis provides totally ordered multicast primitives that guarantee all processes see messages in the same order, a powerful building block for consistent updates of distributed replicas; dOS implements DMP-O to enforce a deterministic order on both implicit shared-memory and explicit OS-channel communications between threads and processes. Unlike dOS, Isis does not guarantee the deterministic execution of a process or the deterministic timing of message delivery relative to processes' instruction sequence. Unlike Isis, dOS does not provide fault tolerance, distributed group membership services, or state transfer to new group members.

8. Conclusions

We introduced the DPG abstraction, which allows programmers to define a deterministic box inside which all communication happens deterministically. We described the shim layer, an interface through which external nondeterminism can be observed and controlled by user-space programs. We developed dOS, an implementation of DPGs in Linux. We demonstrated the shim layer with three applications: record/replay, multithreaded replication, and deterministic filesystem services.

Our evaluation showed that DPGs have reasonable cost in reactive applications such as Apache and Chromium, and also in several parallel workloads. This conceivably enables deterministic execution in deployment, which would fully leverage the benefits of determinism in testing, reliability and debugging.

9. Acknowledgments

We thank Karin Strauss, Dan Grossman, John Zahorjan, and the members of the UW Sampa and systems research groups for their feedback and help. We also thank our anonymous reviewers and our shepherd, Ed Nightingale, for their guidance. This work was supported in part by the National Science Foundation under grants CNS-0627367, CNS-0430477, and CAREER award 0846004, a Torode Family Endowed Career Development Professorship, a Microsoft Faculty Fellowship, and gifts from Nortel Networks and Intel Corporation.

References

[1] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *SOSP*, 2009.

[2] AMD. Advanced Synchronization Facility: Proposed Architectural Specification. `http://developer.amd.com/cpu/ASF/Pages/default.aspx`, March 2009.

[3] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. In *OSDI*, 2010.

[4] C. Basile, Z. Kalbarczyk, and R. Iyer. A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas. In *International Symposium on AUtomated Analysis-driven Debugging*, 2005.

[5] C. Basile, Z. Kalbarczyk, and R. Iyer. Active Replication of Multithreaded Applications. *IEEE Trans. Parallel Distrib. Syst.*, 17(5), 2006.

[6] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, 2010.

[7] E. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe and Efficient Concurrent Programming. In *OOPSLA*, 2009.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[9] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12), December 1993.

[10] G. Blelloch. NESL: A Nested Data-Parallel Language (Version 3.1). Technical report, CMU.

[11] R. Bocchino, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

[12] D. Cahill. NullLogic HTTPd. `http://www.nulllogic.ca/httpd/`.

[13] J. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *SIGMETRICS SPDT*, 1998.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, 2009.

[15] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *OSDI*, 2002.

[16] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution replay of multiprocessor virtual machines. In *VEE*, 2008.

[17] S. A. Edwards and O. Tardieu. SHIM: A Deterministic Model for Heterogeneous Embedded Systems. In *EMSOFT*, 2005.

[18] J. Gilchrist. pbzip2: parallel bzip2. `http://compression.ca/pbzip2`.

[19] Z. Gu, X. W, J. Tang, X. Liu, Z. Xu, M. Wu, F. Kaashoek, and Z. Zhang. R2: An Application-Level Kernel for Record and Replay. In *OSDI*, 2008.

[20] M. Hill and M. Xu. Racey: A Stress Test for Deterministic Execution. `http://www.cs.wisc.edu/~markhill/racey.html`.

[21] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *SIGMETRICS*, 2010.

[22] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Comput.*, 36(4), 1987.

[23] D. Lee, B. Wester, J. Flinn, S. Narayanasamy, and P. Chen. Respec: Efficient Multiprocessor Replay via Speculation and External Determinism. In *ASPLOS*, 2010.

[24] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA*, 2008.

[25] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *ASPLOS*, 2009.

[26] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. *Performance Evaluation Review*, 26(4), 1998.

[27] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, 2008.

[28] P. Narasimhan, L. Moser, and P. Melliar-Smith. Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications. In *Symposium on Reliable Distributed Systems*, 1999.

[29] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.

[30] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design an Implementation of Zap: A System for Migrating Computing Environments. In *OSDI*, 2002.

[31] S. Parka, W. Xiong, Z. Yin, R. Kaushik, K. Lee, S. Lu, and Y. Zhou. Do You Have to Reproduce the Bug at the First Replay Attempt? – PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP*, 2009.

[32] J. Pool, I. Wong, and D. Lie. Relaxed Determinism: Making Redundant Execution on Multiprocessors Practical. In *HotOS*, 2007.

[33] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *ISCA*, 2005.

[34] C. Reis and S. Gribble. Isolating Web Programs in Modern Browser Architectures. In *EuroSys*, 2009.

[35] M. Rinard and M. Lam. The Design, Implementation, and Evaluation of Jade. *ACM TOPLAS*, 20(3), 1988.

[36] M. Ronsse and K. D. Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM TOCS*, 17(2), 1999.

[37] Y. Saito. Jockey: A User-Space Library for Record-Replay Debugging. In *International Symposium on Automated Analysis-driven Debugging*, 2005.

[38] F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4), 1990.

[39] J. Slember and P. Narasimhan. Static Analysis Meets Distributed Fault-Tolerance: Enabling State-Machine Replication With Nondeterminism. In *HotDep*, 2006.

[40] J. Slye and E. Elnozahy. Supporting Nondeterministic Execution in Fault-Tolerant Systems. In *FTCS*, 1996.

[41] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC*, 2002.

[42] R. Van Renesse, K. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4), April 1996.

[43] V. Weaver and S. McKee. Can Hardware Performance Counters be Trusted? In *IISWC*, 2008.

[44] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[45] M. Xu, R. Bodik, and M. Hill. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *ISCA*, 2003.

[46] M. Xu, M. Hill, and R. Bodik. A Regulated Transitive Reduction for Longer Memory Race Recording. In *ASPLOS*, 2006.

[47] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error Diagnosis by Connecting Clues From Run-time Logs. In *ASPLOS*, 2010.

[48] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *EuroSys*, 2010.

Efficient System-Enforced Deterministic Parallelism

Amittai Aviram, Shu-Chun Weng, Sen Hu, Bryan Ford
Yale University

Abstract

Deterministic execution offers many benefits for debugging, fault tolerance, and security. Current methods of executing *parallel* programs deterministically, however, often incur high costs, allow misbehaved software to defeat repeatability, and transform time-dependent races into input- or path-dependent races without eliminating them. We introduce a new parallel programming model addressing these issues, and use Determinator, a proof-of-concept OS, to demonstrate the model's practicality. Determinator's microkernel API provides only "shared-nothing" address spaces and deterministic interprocess communication primitives to make execution of all unprivileged code—well-behaved or not—precisely repeatable. Atop this microkernel, Determinator's user-level runtime adapts optimistic replication techniques to offer a *private workspace* model for both thread-level and process-level parallel programing. This model avoids the introduction of read/write data races, and converts write/write races into reliably-detected conflicts. Coarse-grained parallel benchmarks perform and scale comparably to nondeterministic systems, on both multicore PCs and across nodes in a distributed cluster.

1 Introduction

We often wish to run software *deterministically*, so that from a given input it always produces the same output. Determinism is the foundation of replay debugging [37, 39, 46, 56], fault tolerance [15, 18, 50], and accountability mechanisms [30, 31]. Methods of intrusion analysis [22, 34] and timing channel control [4] further assume the system can *enforce* determinism even on malicious code designed to evade analysis. Executing parallel software deterministically is challenging, however, because threads sharing an address space—or processes sharing resources such as file systems—are prone to nondeterministic, timing-dependent *races* [3, 40, 42, 43].

User-space techniques for parallel deterministic execution [8, 10, 20, 21, 44] show promise but have limitations. First, by relying on a *deterministic scheduler* residing in the application process, they permit buggy or malicious applications to compromise determinism by interfering with the scheduler. Second, deterministic schedulers emulate conventional APIs by synthesizing a repeatable—but arbitrary—schedule of inter-thread interactions, often using an instruction counter as an artificial time metric. Data races remain, therefore, but their manifestation depends subtly on inputs and code path lengths instead of on "real" time. Third, the user-level instrumentation required to isolate and schedule threads' memory accesses can incur considerable overhead, even on coarse-grained code that synchronizes rarely.

To meet the software development, debugging, and security challenges that ubiquitous parallelism presents, it may be insufficient to shoehorn the standard nondeterministic programming model into a synthetic execution schedule. Instead we propose to rethink the basic model itself. We would like a parallel environment that: (a) is "deterministic by default" [12, 40], except when we inject nondeterminism explicitly via external inputs; (b) introduces no data races, either at the memory access level [25, 43] or at higher semantic levels [3]; (c) can enforce determinism on arbitrary, compromised or malicious code for security reasons; and (d) is efficient enough to use for "normal-case" execution of deployed code, not just for instrumentation during development.

As a step toward such a model, we present *Determinator*, a proof-of-concept OS designed around the above goals. Due to its OS-level approach, Determinator supports existing languages, can enforce deterministic execution not only on a single process but on groups of interacting processes, and can prevent malicious user-level code from subverting the kernel's guarantee of determinism. In order to explore the design space freely, Determinator takes a "clean-slate" approach, making few compromises for backward compatibility with existing kernels or APIs. Determinator's programming model could be implemented in a legacy kernel for backward compatibility, however, as part of a "deterministic sandbox" for example [9]. Determinator's user-level runtime also provides limited emulation of the Unix process, thread, and file APIs, to simplify application porting.

Determinator's kernel enforces determinism by denying user code direct access to hardware resources whose use can yield nondeterministic behavior, including real-time clocks, cycle counters, and writable shared memory. Determinator constrains user code to run within a hierarchy of single-threaded, process-like *spaces*, each having a private virtual address space. The kernel's low-level API provides only three system calls, with which a space can synchronize and communicate with its immediate parent and children. Potentially useful sources of nondeterminism, such as timers, Determinator encapsulates into I/O devices, which unprivileged spaces can access

only via explicit communication with more privileged spaces. A supervisory space can thus mediate all nondeterministic inputs affecting a subtree of unprivileged spaces, logging true nondeterministic events for future replay or synthesizing artificial events, for example.

Atop this minimal kernel API, Determinator's user-level runtime emulates familiar shared-resource programming abstractions. The runtime employs file replication and versioning [47] to offer applications a logically shared file system accessed via the Unix file API, and adapts distributed shared memory [2, 17] to emulate shared memory for multithreaded applications. Since this emulation is implemented in user space, applications can freely customize it, and runtime bugs cannot compromise the kernel's guarantee of determinism.

Rather than strictly emulating a conventional, nondeterministic API and consistency model like deterministic schedulers do [8–10, 21, 44], Determinator explores a novel *private workspace* model. In this model, each thread keeps a private virtual replica of all shared memory and file system state; normal reads and writes access and modify this working copy. Threads reconcile their changes only at program-defined synchronization points, much as developers use version control systems. This model eliminates read/write data races, because reads see only *causally prior* writes in the explicit synchronization graph, and write/write races become *conflicts*, which the runtime reliably detects and reports independently of any (real or synthetic) execution schedule.

Experiments with common parallel benchmarks suggest that Determinator can run coarse-grained parallel applications deterministically with both performance and scalability comparable to nondeterministic environments. Determinism incurs a high cost on fine-grained parallel applications, however, due to Determinator's use of virtual memory to isolate threads. For "embarrassingly parallel" applications requiring little inter-thread communication, Determinator can distribute the computation across nodes in a cluster mostly transparently to the application, maintaining usable performance and scalability. As a proof-of-concept, however, the current prototype has many limitations, such as a restrictive space hierarchy, limited file system size, no persistent storage, and inefficient cross-node communication.

This paper makes four main contributions. First, we present the first OS designed from the ground up to offer system-enforced deterministic execution, for both multithreaded processes and groups of interacting processes. Second, we introduce a *private workspace* model for deterministic parallel programming, which eliminates read/write data races and converts schedule-dependent write/write races into reliably-detected, schedule-independent conflicts. Third, we use this model to emulate shared memory and file system abstractions in Determinator's user-space runtime. Fourth, we demonstrate experimentally that this model is practical and efficient enough for "normal-case" use, at least for coarse-grained parallel applications.

Section 2 outlines the deterministic programming model we seek to create. Section 3 then describes the Determinator kernel's design and API, and Section 4 details its user-space application runtime. Section 5 examines our prototype implementation, and Section 6 evaluates it informally and experimentally. Finally, Section 7 outlines related work, and Section 8 concludes.

2 A Deterministic Programming Model

Determinator's basic goal is to offer a programming model that is *naturally* and *pervasively deterministic*. To be *naturally deterministic*, the model's basic abstractions should avoid introducing data races or other nondeterministic behavior in the first place, and not merely provide ways to control, detect, or reproduce races. To be *pervasively deterministic*, the model should behave deterministically at all levels of abstraction: e.g., for shared memory access, inter-thread synchronization, file system access, inter-process communication, external device or network access, and thread/process scheduling.

Intermediate design points are possible and may yield useful tradeoffs. Enforcing determinism only on synchronization and not on low-level memory access might improve efficiency, for example, as in Kendo [44]. For now, however, we explore whether a "purist" approach to pervasive determinism is feasible and practical.

To achieve this goal, we must address timing dependencies in at least four aspects of current systems: in way applications obtain semantically-relevant nondeterministic inputs they require for operation; in shared state such as memory and file systems; in the synchronization APIs threads and processes use to coordinate; and in the namespaces with which applications use and manage system resources. We make no claim that these are the only areas in which current operating systems introduce nondeterminism, but they are the aspects we found essential to address in order to build a working, pervasively deterministic OS. We discuss each area in turn.

2.1 Explicit Nondeterministic Inputs

Many applications use nondeterministic *inputs*, such as incoming messages for a web server, timers for an interactive or real-time application, and random numbers for a cryptographic algorithm. We seek not to eliminate application-relevant nondeterministic inputs, but to make such inputs explicit and controllable.

Mechanisms for parallel debugging [39, 46, 56], fault tolerance [15, 18, 50], accountability [30, 31], and intrusion analysis [22, 34] all rely on the ability to replay a computation instruction-for-instruction, in order to repli-

cate, verify, or analyze a program's execution history. Replay can be efficient when only I/O need be logged, as for a uniprocessor virtual machine [22], but becomes much more costly if *internal* sources of nondeterminism due to parallelism must also be replayed [19, 23].

Determinator therefore transforms useful sources of nondeterminism into explicit I/O, which applications may obtain via controllable channels, and eliminates only internal nondeterminism resulting from parallelism. If an application calls `gettimeofday()`, for example, then a supervising process can intercept this I/O to log, replay, or synthesize these explicit time inputs.

2.2 A Race-Free Model for Shared State

Conventional systems give threads direct, concurrent access to many forms of shared state, such as shared memory and file systems, yielding data races and heisenbugs if the threads fail to synchronize properly [25, 40, 43]. While replay debuggers [37,39,46,56] and deterministic schedulers [8,10,20,21,44] make data races reproducible once they manifest, they do not change the inherently race-prone model in which developers write applications.

Determinator replaces the standard concurrent access model with a *private workspace* model, in which data races do not arise in the first place. This model gives each thread a complete, private virtual replica of all logically shared state a thread may access, including shared memory and file system state. A thread's normal reads and writes affect only its private working state, and do not interact directly with other threads. Instead, Determinator accumulates each threads's changes to shared state, then *reconciles* these changes among threads only at program-defined synchronization points. This model is related to and inspired by early parallel Fortran systems [7, 51], replicated file systems [47], transactional memory [33, 52] and operating systems [48], and distributed version control systems [29], but to our knowledge Determinator is the first OS to introduce a model for pervasive thread- and process-level determinism.

If one thread executes the assignment '$x = y$' while another concurrently executes '$y = x$', for example, these assignments race in the conventional model, but are race-free under Determinator and always swap x with y. Each thread's read of x or y always sees the "old" version of that variable, saved in the thread's private workspace at the last explicit synchronization point.

Figure 1 illustrates a more realistic example of a game or simulator, which uses an array of "actors" (players, particles, etc.) to represent some logical "universe," and updates all of the actors in parallel at each time step. To update the actors, the main thread forks a child thread to process each actor, then synchronizes by joining all these child threads. The child thread code to update each actor is shown "inline" within the **main()** function, which

```
struct actor_state actor[nactors];

main()
  initialize all elements of actor[] array
  for (time = 0; ; time++)
    for (i = 0; i < nactors; i++)
      if (thread_fork(i) == IN_CHILD)
        // child thread to process actor[i]
        examine state of nearby actors
        update state of actor[i] accordingly
        thread_exit();
    for (i = 0; i < nactors; i++)
      thread_join(i);
```

Figure 1: C pseudocode for lock-step time simulation, which contains a data race in standard concurrency models but is bug-free under Determinator.

under Unix works only with process-level `fork()`; Determinator offers this convenience for shared memory threads as well, as discussed later in Section 4.4.

In this example, each child thread reads the "prior" state of any or all actors in the array, then updates the state of its assigned actor "in-place," without any explicit copying or additional synchronization. With standard threads this code has a read/write race: each child thread may see an arbitrary mix of "old" and "new" states as it examines other actors in the array. Under Determinator, however, this code is correct and race-free. Each child thread reads only its private working copy of the actors array, which is untouched (except by the child thread itself) since the main thread forked that child. As the main thread rejoins all its child threads, Determinator merges each child's actor array updates back into the main thread's working copy, for use in the next time step.

While read/write races disappear in Determinator's model, traditional write/write races become *conflicts*. If two child threads concurrently write to the same actor array element, for example, Determinator detects this conflict and signals a runtime exception when the main thread attempts to join the second conflicting child. In the conventional model, by contrast, the threads' execution schedules might cause either of the two writes to "win" and silently propagate its likely erroneous value throughout the computation. Running this code under a conventional deterministic scheduler causes the "winner" to be decided based on a synthetic, reproducible time metric (e.g., instruction count) rather than real time, but the race remains and may still manifest or vanish due to slight changes in inputs or instruction path lengths.

2.3 A Race-Free Synchronization API

Conventional threads can still behave nondeterministically even in a correctly locked program with no low-

level data races. Two threads might acquire a lock in any order, for example, leading to high-level data races [3]. This source of nondeterminism is inherent in the lock abstraction: we can record and replay or synthesize a lock acquisition schedule [44], but such a schedule is still arbitrary and effectively unpredictable to the developer.

Fortunately, many other synchronization abstractions are naturally deterministic, such as fork/join, barriers, and futures [32]. Deterministic abstractions have the key property that when threads synchronize, *program logic alone* determines at what points in the threads' execution paths the synchronization occurs, and which threads are involved. In fork/join synchronization, for example, the parent's **thread_join(**t**)** operation and the child's **thread_exit()** determine the respective synchronization points, and the parent indicates explicitly the thread t to join. Locks fail this test because one thread's **unlock()** passes the lock to an arbitrary successor thread's **lock()**. Queue abstractions such as semaphores and pipes are deterministic if only one thread can access each end of the queue [24, 36], but nondeterministic if several threads can race to insert or remove elements at either end. A related draft elaborates on these considerations [5].

Since the multicore revolution is young and most application code is yet to be parallelized, we may still have a choice of what synchronization abstractions to use. Determinator therefore supports only race-free synchronization primitives natively, although it can emulate nondeterministic primitives via deterministic scheduling for compatibility, as described later in Section 4.5.

2.4 Race-Free System Namespaces

Current operating system APIs often introduce nondeterminism unintentionally by exposing shared namespaces implicitly synchronized by locks. Execution timing affects the pointers returned by `malloc()` or `mmap()` or the file numbers returned by `open()` in multi-threaded Unix processes, and the process IDs returned by `fork()` or the file names returned by `mktemp()` in single-threaded processes. Even if only one thread actually uses a given memory block, file, process ID, or temporary file, the assignment of these names from a shared namespace is inherently nondeterministic.

Determinator's API therefore avoids creating shared namespaces with system-chosen names, instead favoring thread-private namespaces with application-chosen names. Application code, not the system, decides where to allocate memory and what process IDs to assign children. This principle ensures that naming a resource reveals no shared state information other than what the application itself provided. Since implicitly shared namespaces often cause multiprocessor contention, designing system APIs to avoid this implicit sharing may be synergistic with recent multicore scalability work [14].

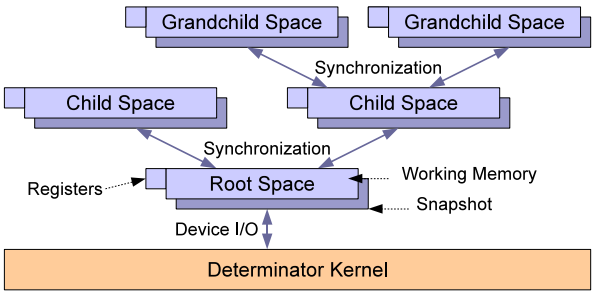


Figure 2: The kernel's hierarchy of *spaces*, each containing private register and virtual memory state.

3 The Determinator Kernel

Having outlined the principles underlying Determinator's programming model, we now describe its kernel design. Normal applications do not use the kernel API directly, but rather the higher-level abstractions provided by the user-level runtime, described in the next section. We make no claim that our kernel design or API is the "right" design for a determinism-enforcing kernel, but merely that it illustrates one way to implement a pervasively deterministic application environment.

3.1 Spaces

Determinator executes application code within an arbitrarily deep hierarchy of *spaces*, illustrated in Figure 2. Each space consists of CPU register state for a single control flow, and private virtual memory containing code and data directly accessible within that space. A Determinator space is analogous to a single-threaded Unix process, with important differences; we use the term "space" to highlight these differences and avoid confusion with the "process" and "thread" abstractions Determinator emulates at user level, described in Section 4.

As in a nested process model [27], a Determinator space cannot outlive its parent, and a space can directly interact *only* with its immediate parent and children via three system calls described below. The kernel provides no file systems, writable shared memory, or other abstractions that imply globally shared state.

Only the distinguished *root space* has direct access to nondeterministic inputs via I/O devices, such as console input or clocks. Other spaces can access I/O devices only indirectly via parent/child interactions, or via I/O privileges delegated by the root space. A parent space can thus control all nondeterministic inputs into any unprivileged space subtree, e.g., logging inputs for future replay. This space hierarchy also creates a performance bottleneck for I/O-bound applications, a limitation of the current design we intend to address in future work.

Call	Interacts with	Description
Put	Child space	Copy register state and/or virtual memory range into child, and optionally start child executing.
Get	Child space	Copy register state, virtual memory range, and/or changes since the last snapshot out of a child.
Ret	Parent space	Stop and wait for parent to issue a Get or Put. Processor traps also cause implicit Ret.

Table 1: System calls comprising Determinator's kernel API.

Put	Get	Option	Description
✓	✓	Regs	PUT/GET child's register state.
✓	✓	Copy	Copy memory to/from child.
✓	✓	Zero	Zero-fill virtual memory range.
✓		Snap	Snapshot child's virtual memory.
✓		Start	Start child space executing.
	✓	Merge	Merge child's changes into parent.
✓	✓	Perm	Set memory access permissions.
✓	✓	Tree	Copy (grand)child subtree.

Table 2: Options/arguments to the Put and Get calls.

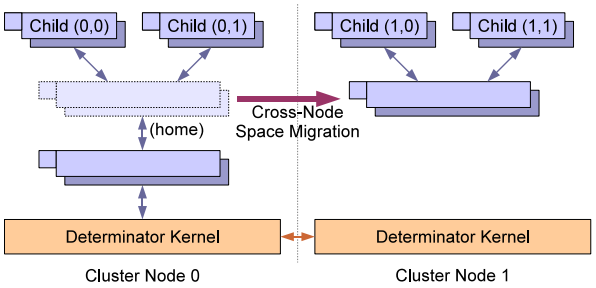


Figure 3: A spaces migrating among two nodes and starting child spaces on each node.

3.2 System Call API

Determinator spaces interact only as a result of processor traps and the kernel's three system calls—Put, Get, and Ret, summarized in Table 1. Put and Get take several optional arguments, summarized in Table 2. Most options can be combined: e.g., in one Put call a space can initialize a child's registers, copy a range of the parent's virtual memory into the child, set page permissions on the destination range, save a complete snapshot of the child's address space, and start the child executing.

Each space has a private namespace of child spaces, which user-level code manages. A space specifies a child number to Get or Put, and the kernel creates that child if it doesn't already exist, before performing the requested operations. If the specified child did exist and was still executing at the time of the Put/Get call, the kernel blocks the parent's execution until the child stops due to a Ret system call or a processor trap. These "rendezvous" semantics ensure that spaces synchronize only at well-defined points in both spaces' execution.

The Copy option logically copies a range of virtual memory between the invoking space and the specified child. The kernel uses copy-on-write to optimize large copies and avoid physically copying read-only pages.

Merge is available only on Get calls. A Merge is like a Copy, except the kernel copies only bytes that *differ* between the child's current and reference snapshots into the parent space, leaving other bytes in the parent untouched. The kernel also detects conflicts: if a byte changed in *both* the child's and parent's spaces since the snapshot, the kernel generates an exception, treating a conflict as a programming error like an illegal memory access or divide-by-zero. Determinator's user-level runtime uses Merge to give multithreaded processes the illusion of shared memory, as described later in Section 4.4. In principle, user-level code could implement Merge itself, but the kernel's direct access to page tables makes it easy for the kernel to implement Merge efficiently.

Finally, the Ret system call stops the calling space, returning control to the space's parent. Exceptions such as divide-by-zero also cause a Ret, providing the parent a status code indicating why the child stopped.

To facilitate debugging and prevent untrusted children from looping forever, a parent can start a child with an *instruction limit*, forcing control back to the parent after the child and its descendants collectively execute this many instructions. Counting instructions instead of "real time" preserves determinism, while enabling spaces to "quantize" a child's execution to implement scheduling schemes deterministically at user level [8, 21].

Barring kernel or processor bugs, unprivileged spaces constrained to use the above kernel API alone cannot behave nondeterministically even by deliberate design. While a formal proof is out of scope, one straightforward argument is that the above Get/Put/Ret primitives reduce to blocking, one-to-one message channels, making the space hierarchy a deterministic Kahn network [36].

3.3 Distribution via Space Migration

The kernel allows space hierarchies to span not only multiple CPUs in a multiprocessor/multicore system, but also multiple nodes in a homogeneous cluster, mostly transparently to application code. While distribution is semantically transparent to applications, an application may have to be designed with distribution in mind to perform well. As with other aspects of the kernel's design, we make no pretense that this is the "right" approach to cross-node distribution, but merely one way to extend a deterministic execution model across a cluster.

Distribution adds no new system calls or options to the API above. Instead, the Determinator kernel inter-

prets the higher-order bits in each process's child number namespace as a "node number" field. When a space invokes Put or Get, the kernel first logically migrates the calling space's state and control flow to the node whose number the user specifies as part of its child number argument, before creating and/or interacting with some child on that node, as specified in the remaining child number bits. Figure 3 illustrates a space migrating between two nodes and managing child spaces on each.

Once created, a space has a *home node*, to which the space migrates when interacting with its parent on a Ret or trap. Nodes are numbered so that "node zero" in any space's child namespace always refers to the space's home node. If a space uses only the low bits in its child numbers and leaves the node number field zero, the space's children all have the same home as the parent.

When the kernel migrates a space, it first transfers to the receiving kernel only the space's register state and address space summary information. Next, the receiving kernel requests the space's memory pages on demand as the space accesses them on the new node. Each node's kernel avoids redundant cross-node page copying in the common case when a space repeatedly migrates among several nodes—e.g., when a space starts children on each of several nodes, then returns later to collect their results. For pages that the migrating space only reads and never writes, such as program code, each kernel reuses cached copies of these pages whenever the space returns to that node. The kernel currently performs no prefetching or other adaptive optimizations. Its rudimentary messaging protocol runs directly atop Ethernet, and does not support TCP/IP for Internet-wide distribution.

4 Emulating High-Level Abstractions

Determinator's kernel API eliminates many convenient and familiar abstractions; can we reproduce them under strict determinism? We find that many familiar abstractions remain feasible, though with important tradeoffs. This section describes how Determinator's user-level runtime infrastructure emulates traditional Unix processes, file systems, threads, and synchronization.

4.1 Processes and fork/exec/wait

We make no attempt to replicate Unix process semantics exactly, but would like to emulate traditional `fork`/`exec`/`wait` APIs enough to support common uses in scriptable shells, build tools, and multi-process "batch processing" applications such as compilers.

Fork: Implementing a basic Unix `fork()` requires only one Put system call, to copy the parent's entire memory state into a child space, set up the child's registers, and start the child. The difficulty arises from Unix's global process ID (PID) namespace, a source of nondeterminism as discussed in Section 2.4. Since most applications use PIDs returned by `fork()` merely as an opaque argument to a subsequent `waitpid()`, our runtime makes PIDs local to each process: one process's PIDs are unrelated to, and may numerically conflict with, PIDs in other processes. This change breaks Unix applications that pass PIDs among processes, and means that commands like '`ps`' must be built into shells for the same reason that '`cd`' already is. This simple approach works for compute-oriented applications following the typical fork/wait pattern, however.

Since `fork()` returns a PID chosen by the system, while our kernel API requires user code to manage child numbers, our user-level runtime maintains a "free list" of child spaces and reserves one during each `fork()`. To emulate Unix process semantics more closely, a central space such as the root space could manage a global PID namespace, at the cost of requiring inter-space communication during operations such as `fork()`.

Exec: A user-level implementation of Unix `exec()` must construct the new program's memory image, intended to replace the old program, while still executing the old program's runtime library code. Our runtime loads the new program into a "reserved" child space never used by `fork()`, then calls Get to copy that child's entire memory atop that of the (running) parent: this Get thus "returns" into the new program. To ensure that the instruction address following the old program's Get is a valid place to start the new program, the runtime places this Get in a small "trampoline" code fragment mapped at the same location in the old and new programs. The runtime also carries over some Unix process state, such as the the PID namespace and file system state described later, from the old to the new program.

Wait: When an application calls `waitpid()` to wait for a specific child, the runtime calls Get to synchronize with the child's Ret and obtain the child's exit status. The child may return to the parent before terminating, in order to make I/O requests as described below; in this case, the parent's runtime services the I/O request and resumes the `waitpid()` transparently to the application.

Unix's `wait()` is more challenging, as it waits for *any* (i.e., "the first") child to terminate, violating the constraints of deterministic synchronization discussed in Section 2.3. Our kernel's API provides no system call to "wait for any child," and can't (for unprivileged spaces) without compromising determinism. Instead, our runtime waits for the child that was forked earliest whose status was not yet collected.

This behavior does not affect applications that fork one or more children and then wait for all of them to complete, but affects two common uses of `wait()`. First, interactive Unix shells use `wait()` to report when back-

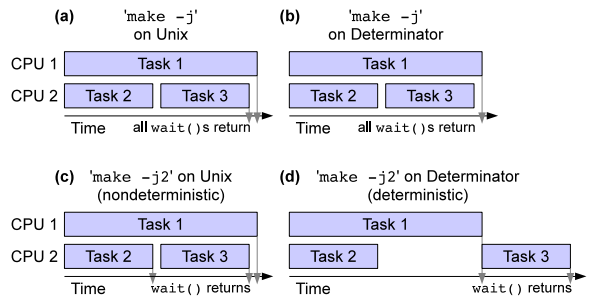


Figure 4: Example parallel `make` scheduling scenarios under Unix versus Determinator: (a) and (b) with unlimited parallelism (no user-level scheduling); (c) and (d) with a "2-worker" quota imposed at user level.

ground processes complete; thus, an interactive shell running under Determinator requires special "nondeterministic I/O privileges" to provide this functionality (and related functions such as interactive job control). Second, our runtime's behavior may adversely affect the performance of programs that use `wait()` to implement dynamic scheduling or load balancing in user space.

Consider a parallel `make` run with or without limiting the number of concurrent children. A plain '`make -j`', allowing unlimited children, leaves scheduling decisions to the system. Under Unix or Determinator, the kernel's scheduler dynamically assigns tasks to available CPUs, as illustrated in Figure 4 (a) and (b). If the user runs '`make -j2`', however, then `make` initially starts only tasks 1 and 2, then waits for one of them to complete before starting task 3. Under Unix, `wait()` returns when the short task 2 completes, enabling `make` to start task 3 immediately as in (c). On Determinator, however, the `wait()` returns only when (deterministically chosen) task 1 completes, resulting in a non-optimal schedule (d): determinism prevents the runtime from learning which of tasks 1 and 2 completed first. The unavailability of timing information with which to make good application-level scheduling decisions thus suggests a practice of leaving scheduling to the system in a deterministic environment (e.g., '`make -j`' instead of '`-j2`').

4.2 A Shared File System

Unix's globally shared file system provides a convenient namespace and repository for staging program inputs, storing outputs, and holding intermediate results such as temporary files. Since our kernel permits no physical state sharing, user-level code must emulate shared state abstractions. Determinator's "shared-nothing" space hierarchy is similar to a distributed system consisting only of uniprocessor machines, so our user-level runtime borrows distributed file system principles to offer applications a shared file system abstraction.

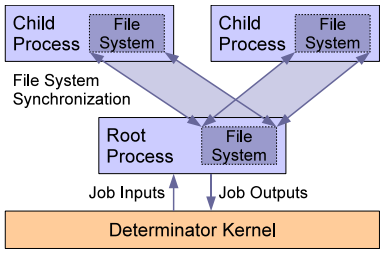


Figure 5: Each user-level runtime maintains a private replica of a logically shared file system, using file versioning to reconcile replicas at synchronization points.

Since our current focus is on emulating familiar abstractions and not on developing storage systems, Determinator's file system currently provides no persistence: it effectively serves only as a temporary file system.

While many distributed file system designs may be applicable, our runtime uses replication with weak consistency [53, 55]. Our runtime maintains a complete file system replica in the address space of each process it manages, as shown in Figure 5. When a process creates a child via `fork()`, the child inherits a copy of the parent's file system in addition to the parent's open file descriptors. Individual `open`/`close`/`read`/`write` operations in a process use only that process's file system replica, so different processes' replicas may diverge as they modify files concurrently. When a child terminates and its parent collects its state via `wait()`, the parent's runtime copies the child's file system image into a scratch area in the parent space and uses file versioning [47] to propagate the child's changes into the parent.

If a shell or parallel `make` forks several compiler processes in parallel, for example, each child writes its output `.o` file to its own file system replica, then the parent's runtime merges the resulting `.o` files into the parent's file system as the parent collects each child's exit status. This copying and reconciliation is not as inefficient as it may appear, due to the kernel's copy-on-write optimizations. Replicating a file system image among many spaces copies no physical pages until user-level code modifies them, so all processes' copies of identical files consume only one set of pages.

As in any weakly-consistent file system, processes may cause *conflicts* if they perform unsynchronized, concurrent writes to the same file. When our runtime detects a conflict, it simply discards one copy and sets a conflict flag on the file; subsequent attempts to `open()` the file result in errors. This behavior is intended for batch compute applications for which conflicts indicate an application or build system bug, whose appropriate solution is to fix the bug and re-run the job. Interactive use would demand a conflict handling policy that avoids losing data. The user-level runtime could alternatively use

pessimistic locking to implement stronger consistency and avoid unsynchronized concurrent writes, at the cost of more inter-space communication.

The current design's placement of each process's file system replica in the process's own address space has two drawbacks. First, it limits total file system size to less than the size of an address space; this is a serious limitation in our 32-bit prototype, though it may be less of an issue on a 64-bit architecture. Second, wild pointer writes in a buggy process may corrupt the file system more easily than in Unix, where a buggy process must actually call `write()` to corrupt a file. The runtime could address the second issue by write-protecting the file system area between calls to `write()`, or it could address both issues by storing file system data in child spaces not used for executing child processes.

4.3 Input/Output and Logging

Since unprivileged spaces can access external I/O devices only indirectly via parent/child interaction within the space hierarchy, our user-level runtime treats I/O as a special case of file system synchronization. In addition to regular files, a process's file system image can contain special *I/O files*, such as a console input file and a console output file. Unlike Unix device special files, Determinator's I/O files actually hold data in the process's file system image: for example, a process's console input file accumulates all the characters the process has received from the console, and its console output file contains all the characters it has written to the console. In the current prototype this means that console or log files can eventually "fill up" and become unusable, though a suitable garbage-collection mechanism could address this flaw.

When a process does a `read()` from the console, the C library first returns unread data already in the process's local console input file. When no more data is available, instead of returning an end-of-file condition, the process calls Ret to synchronize with its parent and wait for more console input (or in principle any other form of new input) to become available. When the parent does a `wait()` or otherwise synchronizes with the child, it propagates any new input it already has to the child. When the parent has no new input for any waiting children, it forwards all their input requests to its parent, and ultimately to the kernel via the root process.

When a process does a console `write()`, the runtime appends the new data to its internal console output file as it would append to a regular file. The next time the process synchronizes with its parent, file system reconciliation propagates these writes toward the root process, which forwards them to the kernel's I/O devices. A process can request immediate synchronization and output propagation by explicitly calling `fsync()`.

The reconciliation mechanism handles "append-only" writes differently from other file changes, enabling concurrent writes to console or log files without conflict. During reconciliation, if both the parent and child have made append-only writes to the same file, reconciliation appends the child's latest writes to the parent's copy of the file, and vice versa. Each process's replica thus accumulates all processes' concurrent writes, though different processes may observe these writes in a different order. Unlike Unix, rerunning a parallel computation from the same inputs with and without output redirection yields byte-for-byte identical console and log file output.

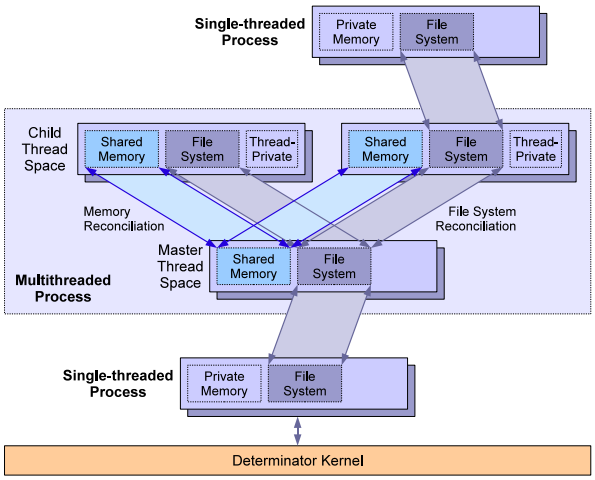


Figure 6: A multithreaded process built from one space per thread, with a master space managing synchronization and memory reconciliation.

4.4 Shared Memory Multithreading

Shared memory multithreading is popular despite the nondeterminism it introduces into processes, in part because parallel code need not pack and unpack messages: threads simply compute "in-place" on shared variables and structures. Since Determinator gives user spaces no physically shared memory other than read-only sharing via copy-on-write, emulating shared memory involves distributed shared memory (DSM) techniques. Adapting the private workspace model discussed in Section 2.2 to thread-level shared memory involves reusing ideas explored in early parallel Fortran machines [7, 51] and in release-consistent DSM systems [2, 17], although none of this prior work attempted to provide determinism.

Our runtime uses the kernel's Snap and Merge operations (Section 3.2) to emulate shared memory in the private workspace model, using fork/join synchronization. To fork a child, the parent thread calls Put with the Copy, Snap, Regs, and Start options to copy the shared part of its memory into a child space, save a snapshot of that memory state in the child, and start the child running, as illustrated in Figure 6. The master thread may fork mul-

tiple children this way. To synchronize with a child and collect its results, the parent calls Get with the Merge option, which merges all changes the child made to shared memory, since its snapshot was taken, back into the parent. If both parent and child—or the child and other children whose changes the parent has collected—have concurrently modified the same byte since the snapshot, the kernel detects and reports this conflict.

Our runtime also supports barriers, the foundation of data-parallel programming models like OpenMP [45]. When each thread in a group arrives at a barrier, it calls Ret to stop and wait for the parent thread managing the group. The parent calls Get with Merge to collect each child's changes before the barrier, then calls Put with Copy and Snap to resume each child with a new shared memory snapshot containing all threads' prior results. While our private workspace model conceptually extends to non-hierarchical synchronization [5], our prototype's strict space hierarchy currently limits synchronization flexibility, an issue we intend to address in the future. *Any* synchronization abstraction may be emulated at some cost as described in the next section, however.

An application can choose which parts of its address space to share and which to keep thread-private. By placing thread stacks outside the shared region, all threads can reuse the same stack area, and the kernel wastes no effort merging stack data. Thread-private stacks also offer the convenience of allowing a child thread to inherit its parent's stack, and run "inline" in the same C/C++ function as its parent, as in Figure 1. If threads wish to pass pointers to stack-allocated structures, however, then they may locate their stacks in disjoint shared regions. Similarly, if the file system area is shared, then the threads share a common file descriptor namespace as in Unix. Excluding the file system area from shared space and using normal file system reconciliation (Section 4.2) to synchronize it yields thread-private file tables.

4.5 Emulating Legacy Thread APIs

As discussed in Section 2.3, we hope much existing sequential code can readily be parallelized using naturally deterministic synchronization abstractions, like data-parallel models such as OpenMP [45] and SHIM [24] already offer. For code already parallelized using nondeterministic synchronization, however, Determinator's runtime can emulate the standard pthreads API via deterministic scheduling [8, 10, 21], at certain costs.

In a process that uses nondeterministic synchronization, the process's initial *master space* never runs application code directly, but instead acts as a deterministic scheduler. This scheduler creates one child space to run each application thread. The scheduler runs the threads under an artificial execution schedule, emulating a schedule by which a true shared-memory multiprocessor might in principle run them, but using a deterministic, virtual notion of time—namely, number of instructions executed—to schedule all inter-thread interactions.

Like DMP [8, 21], our deterministic scheduler *quantizes* each thread's execution by preempting it after executing a fixed number of instructions. Whereas DMP implements preemption by instrumenting user-level code, our scheduler uses the kernel's instruction limit feature (Section 3.2). The scheduler "donates" execution quanta to threads round-robin, allowing each thread to run concurrently with other threads for one quantum, before collecting the thread's shared memory changes via Merge and restarting it for another quantum.

A thread's shared memory writes propagate to other threads only at the end of each quantum, violating sequential consistency [38]. Like DMP-B [8], our scheduler implements a weak consistency model [28], totally ordering only synchronization operations. To enforce this total order, each synchronization operation could simply spin for a a full quantum. To avoid wasteful spinning, however, our synchronization primitives interact with the deterministic scheduler directly.

Each mutex, for example, is always "owned" by some thread, whether or not the mutex is locked. The mutex's owner can lock and unlock the mutex without scheduler interactions, but any other thread needing the mutex must first invoke the scheduler to obtain ownership. At the current owner's next quantum, the scheduler "steals" the mutex from its current owner if the mutex is unlocked, and otherwise places the locking thread on the mutex's queue to be awoken once the mutex becomes available.

Since the scheduler can preempt threads at any point, a challenge common to any preemptive scenario is making synchronization functions such as `pthread_mutex_lock()` atomic. The kernel does not allow threads to disable or extend their own instruction limits, since we wish to use instruction limits at process level as well, e.g., to enforce deterministic "time" quotas on untrusted processes, or to improve user-level process scheduling (see Section 4.1) by quantizing process execution. After synchronizing with a child thread, therefore, the master space checks whether the instruction limit preempted a synchronization function, and if so, resumes the preempted code in the master space. Before returning to the application, these functions check whether they have been "promoted" to the master space, and if so migrate their register state back to the child thread and restart the scheduler in the master space.

While deterministic scheduling provides compatibility with existing parallel code, it has drawbacks. The master space, required to enforce a total order on synchronization operations, may be a scaling bottleneck unless execution quanta are large. Since threads can interact only at quanta boundaries, however, large quanta increase the

time one thread may waste waiting to interact with another, to steal an unlocked mutex for example.

Further, since the deterministic scheduler may preempt a thread and propagate shared memory changes at any point in application code, the *programming model* remains nondeterministic. In contrast with our private workspace model, if one thread runs '$x = y$' while another runs '$y = x$' under the deterministic scheduler, the result may be repeatable but is no more predictable to the programmer than on traditional systems. While rerunning a program with *exactly* identical inputs will yield identical results, if the input is perturbed to change the length of any instruction sequence, these changes may cascade into a different execution schedule and trigger schedule-dependent if not timing-dependent bugs.

5 Prototype Implementation

Determinator is written in C with small assembly fragments, currently runs on the 32-bit x86 architecture, and implements the kernel API and user-level runtime facilities described above. Source releases are available at '`http://dedis.cs.yale.edu/`'.

Since our focus is on parallel compute-bound applications, Determinator's I/O capabilities are currently limited. The system provides text-based console I/O and a Unix-style shell supporting redirection and both scripted and interactive use. The shell offers no interactive job control, which would require currently unimplemented "nondeterministic privileges" (Section 4.1). The system has no demand paging or persistent disk storage: the user-level runtime's logically shared file system abstraction currently operates in physical memory only.

The kernel supports application-transparent space migration among up to 32 machines in a cluster, as described in Section 3.3. Migration uses a synchronous messaging protocol with only two request/response types and implements almost no optimizations such as page prefetching. The protocol runs directly atop Ethernet, and is not intended for Internet-wide distribution.

The prototype has other limitations already mentioned. The kernel's strict space hierarchy could bottleneck I/O-intensive applications (Section 3.1), and does not easily support non-hierarchical synchronization such as queues or futures (Section 4.4). The file system's size is constrained to a process's address space (Section 4.2), and special I/O files can fill up (Section 4.3). None of these limitations are fundamental to Determinator's programming model. At some cost in complexity, the model could support non-hierarchical synchronization [5]. The runtime could store files in child spaces or on external I/O devices, and could garbage-collect I/O streams.

Implementing instruction limits (Section 3.2) requires the kernel to recover control after a precise number of instructions execute in user mode. While the PA-RISC architecture provided this feature [1], the x86 does not, so we borrowed ReVirt's technique [22]. We first set an *imprecise* hardware performance counter, which unpredictably overshoots its target a small amount, to interrupt the CPU before the desired number of instructions, then run the remaining instructions under debug tracing.

6 Evaluation

This section evaluates the Determinator prototype, first informally, then examining single-node and distributed parallel processing performance, and finally code size.

6.1 Experience Using the System

We find that a deterministic programming model simplifies debugging of both applications and user-level runtime code, since user-space bugs are always reproducible. Conversely, when we do observe nondeterministic behavior, it can result only from a kernel (or hardware) bug, immediately limiting the search space.

Because Determinator's file system holds a process's output until the next synchronization event (often the process's termination), each process's output appears as a unit even if the process executes in parallel with other output-generating processes. Further, different processes' outputs appear in a consistent order across runs, as if run sequentially. (The kernel provides a system call for debugging that outputs a line to the "real" console immediately, reflecting true execution order, but chaotically interleaving output as in conventional systems.)

While race detection tools exist [25, 43], we found it convenient that Determinator always detects conflicts under "normal-case" execution, without requiring the user to run a special tool. Since the kernel detects shared memory conflicts and the user-level runtime detects file system conflicts at every synchronization event, Determinator's model makes conflict detection as standard as detecting division by zero or illegal memory accesses.

A subset of Determinator doubles as *PIOS*, "Parallel Instructional Operating System," which we used in Yale's operating system course this spring. While the OS course's objectives did not include determinism, they included introducing students to parallel, multicore, and distributed operating system concepts. For this purpose, we found Determinator/PIOS to be a useful instructional tool due to its simple design, minimal kernel API, and adoption of distributed systems techniques within and across physical machines. PIOS is partly derived from MIT's JOS [35], and includes a similar instructional framework where students fill in missing pieces of a "skeleton." The twelve students who took the course, working in groups of two or three, all successfully reimplemented Determinator's core features: multiprocessor scheduling with Get/Put/Ret coordination, virtual memory with copy-on-write and Snap/Merge, user-level

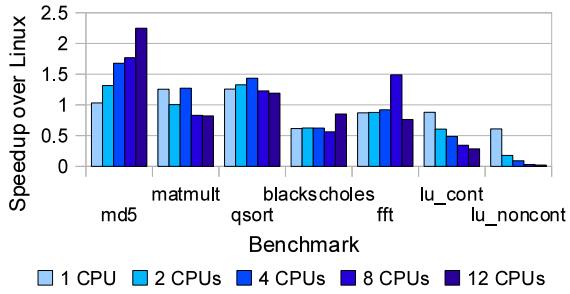


Figure 7: Determinator performance relative to pthreads under Ubuntu Linux on various parallel benchmarks.

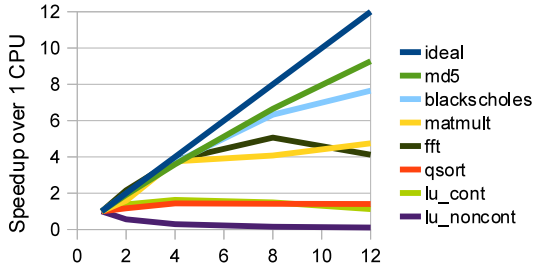


Figure 8: Determinator parallel speedup over its own single-CPU performance on various benchmarks.

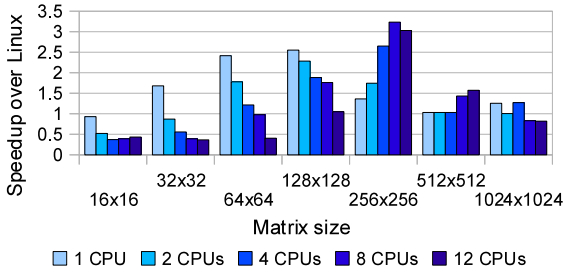


Figure 9: Matrix multiply with varying matrix size.

threads with fork/join synchronization (but not deterministic scheduling), the user-space file system with versioning and reconciliation, and application-transparent cross-node distribution via space migration. In their final projects they extended the OS with features such as graphics, pipes, and a remote shell. While instructional use by no means indicates a system's real-world utility, we find the success of the students in understanding and building on Determinator's architecture promising.

6.2 Single-node Multicore Performance

Since Determinator runs user-level code "natively" on the hardware instead of rewriting user code [8, 21], we expect it to perform comparably to conventional systems when executing single-threaded, compute-bound code. Since thread interactions require system calls, context switches, and virtual memory operations, however, we expect determinism to incur a performance cost in proportion to the frequency of thread interaction.

Figure 7 shows the performance of several shared-memory parallel benchmarks we ported to Determinator, relative to the same benchmarks using conventional pthreads on 32-bit Ubuntu Linux 9.10. The *md5* benchmark searches for an ASCII string yielding a particular MD5 hash, as in a brute-force password cracker; *matmult* multiplies two 1024×1024 integer matrices; *qsort* is a recursive parallel quicksort on an integer array; *blackscholes* is a financial benchmark from the PARSEC suite [11]; and *fft*, *lu_cont*, and *lu_noncont* are Fast Fourier Transform and LU-decomposition benchmarks from SPLASH-2 [57]. We tested all benchmarks on a 2 socket $\times$ 6 core, 2.2GHz AMD Opteron PC.

Coarse-grained benchmarks like *md5*, *matmult*, *qsort*, *blackscholes*, and *fft* show performance comparable with that of nondeterministic multithreaded execution under Linux. The *md5* benchmark shows better scaling on Determinator than on Linux, achieving a $2.25\times$ speedup over Linux on 12 cores. We have not identified the precise cause of this speedup over Linux but suspect scaling bottlenecks in Linux's thread system [54].

Porting the *blackscholes* benchmark to Determinator required no changes as it uses deterministically scheduled pthreads (Section 4.5). The deterministic scheduler's quantization, however, incurs a fixed performance cost of about 35% for the chosen quantum of 10 million instructions. We could reduce this overhead by increasing the quantum, or eliminate it by porting the benchmark to Determinator's "native" parallel API.

The fine-grained *lu* benchmarks show a higher performance cost, indicating that Determinator's virtual memory-based approach to enforcing determinism is not well-suited to fine-grained parallel applications. Future hardware enhancements might make determinism practical for fine-grained parallel applications, however [21].

Figure 8 shows each benchmark's speedup relative to single-threaded execution on Determinator. The "embarrassingly parallel" *md5* and *blackscholes* scale well, *matmult* and *fft* level off after four processors (but still perform comparably to Linux as Figure 7 shows), and the remaining benchmarks scale poorly.

To quantify further the effect of parallel interaction granularity on deterministic execution performance, Figures 9 and 10 show Linux-relative performance of *matmult* and *qsort*, respectively, for varying problem sizes. With both benchmarks, deterministic execution incurs a high performance cost on small problem sizes requiring frequent interaction, but on large problems Determinator is competitive with and sometimes faster than Linux.

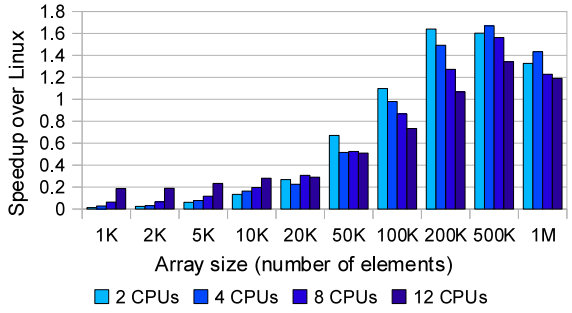


Figure 10: Parallel quicksort with varying array size.

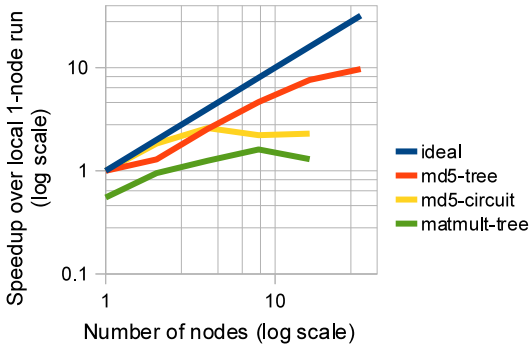


Figure 11: Speedup of deterministic shared memory benchmarks on varying-size distributed clusters.

6.3 Distributed Computing Performance

While Determinator's rudimentary space migration (Section 3.3) is far from providing a full cluster computing architecture, we would like to test whether such a mechanism can extend a deterministic computing model across nodes with usable performance at least for some applications. We therefore changed the *md5* and *matmult* benchmarks to distribute workloads across a cluster of up to 32 uniprocessor nodes via space migration. Both benchmarks still run in a (logical) shared memory model via Snap/Merge. Since we did not have a cluster on which we could run Determinator natively, we ran it under QEMU [6], on a cluster of 2 socket × 2 core, 2.4GHz Intel Xeon machines running SuSE Linux 11.1.

Figure 11 shows parallel speedup under Determinator relative to local single-node execution in the same environment, on a log-log scale. In *md5-circuit*, the master space acts like a traveling salesman, migrating serially to each "worker" node to fork child processes, then retracing the same circuit to collect their results. The *md5-tree* variation forks workers recursively in a binary tree: the master space forks children on two nodes, those children each fork two children on two nodes, etc. The *matmult-tree* benchmark implements matrix multiply with recursive work distribution as in *md5-tree*.

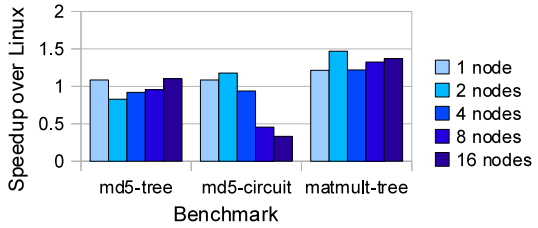


Figure 12: Deterministic shared memory benchmarks versus distributed-memory equivalents for Linux.

Component	Determinator Semicolons	PIOS Semicolons
Kernel core	2044	1847
Hardware/device drivers	751	647
User-level runtime	2952	1079
Generic C library code	6948	394
User-level programs	1797	1418
Total	14,492	5385

Table 3: Implementation code size of the Determinator OS and of PIOS, its instructional subset.

The "embarrassingly parallel" *md5-tree* performs and scales well, but only with recursive work distribution. Matrix multiply levels off at two nodes, due to the amount of matrix data the kernel transfers across nodes via its simplistic page copying protocol, which currently performs no data streaming, prefetching, or delta compression. The slowdown for 1-node distributed execution in *matmult-tree* reflects the cost of transferring the matrix to a (single) remote machine for processing.

As Figure 12 shows, the shared memory *md5-tree* and *matmult-tree* benchmarks, running on Determinator, perform comparably to nondeterministic, distributed-memory equivalents running on Puppy Linux 4.3.1, in the same QEMU environment. Determinator's clustering protocol does not use TCP as the Linux-based benchmarks do, so we explored the benchmarks' sensitivity to this factor by implementing TCP-like round-trip timing and retransmission behavior in Determinator. These changes resulted in less than a 2% performance impact.

Illustrating the simplicity benefits of Determinator's shared memory thread API, the Determinator version of *md5* is 63% the size of the Linux version (62 lines containing semicolons versus 99), which uses remote shells to coordinate workers. The Determinator version of *matmult* is 34% the size of its Linux equivalent (90 lines versus 263), which passes data explicitly via TCP.

6.4 Implementation Complexity

To provide a feel for implementation complexity, Table 3 shows source code line counts for Determinator, as well as its PIOS instructional subset, counting only lines con-

taining semicolons. The entire system is less than 15,000 lines, about half of which is generic C and math library code needed mainly for porting Unix applications easily.

7 Related Work

Recognizing the benefits of determinism [12, 40], parallel languages such as SHIM [24] and DPJ [12, 13] enforce determinism at language level, but require rewriting, rather than just parallelizing, existing serial code. Race detectors [25, 43] detect low-level heisenbugs in nondeterministic parallel programs, but may miss higher-level heisenbugs [3]. Language extensions can dynamically check determinism assertions [16, 49], but heisenbugs may persist if the programmer omits an assertion.

Early parallel Fortran systems [7, 51], release consistent DSM [2, 17], transactional memory [33, 52] and OS APIs [48], replicated file systems [53, 55], and distributed version control [29] all foreshadow Determinator's private workspace programming model. None of these precedents create a deterministic application programming model, however, as is Determinator's goal.

Deterministic schedulers such as DMP [8, 21] and Grace [10] instrument an application to schedule inter-thread interactions on a repeatable, artificial time schedule. DMP isolates threads via code rewriting, while Grace uses virtual memory as in Determinator. Developed simultaneously with Determinator, dOS [9] incorporates a deterministic scheduler into the Linux kernel, preserving Linux's existing programming model and API. This approach provides greater backward compatibility than Determinator's clean-slate design, but makes the Linux programming model no more *semantically* deterministic than before. Determinator offers new thread and process models redesigned to eliminate conventional data races, while supporting deterministic scheduling in user space for backward compatibility.

Many techniques are available to log and replay nondeterministic events in parallel applications [39, 46, 56]. SMP-ReVirt can log and replay a multiprocessor virtual machine [23], supporting uses such as system-wide intrusion analysis [22,34] and replay debugging [37]. Logging a parallel system's nondeterministic events is costly in performance and storage space, however, and usually infeasible for "normal-case" execution. Determinator demonstrates the feasibility of providing system-enforced determinism for normal-case execution, without internal event logging, while maintaining performance comparable with current systems at least for coarse-grained parallel applications.

Determinator's kernel design owes much to microkernels such as L3 [41]. An interesting contrast is with the Exokernel approach [26], which is incompatible with Determinator's. System-enforced determinism requires hiding nondeterministic kernel state from applications, such as the physical addresses of virtual memory pages, whereas exokernels deliberately expose this state.

8 Conclusion

While Determinator is only a proof-of-concept, it shows that operating systems can offer a pervasively and naturally deterministic application environment, avoiding the introduction of data races in shared memory and file system access, thread and process synchronization, and throughout the API. Our experiments suggest that such an environment can efficiently run coarse-grained parallel applications, both on a single multicore machine and across a cluster, though supporting fine-grained parallelism efficiently may require hardware evolution.

Acknowledgments

We thank Zhong Shao, Ramakrishna Gummadi, Frans Kaashoek, Nickolai Zeldovich, Sam King, and the OSDI reviewers for their valuable feedback. We also thank NSF for their support under grant CNS-1017206.

References

[1] *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. Hewlett-Packard, Feb. 1994.

[2] C. Amza et al. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[3] C. Artho, K. Havelund, and A. Biere. High-level data races. In *VVEIS*, pages 82–93, Apr. 2003.

[4] A. Aviram et al. Determinating timing channels in compute clouds. In *CCSW*, Oct. 2010.

[5] A. Aviram and B. Ford. Deterministic consistency, Feb. 2010. `http://arxiv.org/abs/0912.0926`.

[6] F. Bellard. QEMU, a fast and portable dynamic translator, Apr. 2005.

[7] M. Beltrametti, K. Bobey, and J. R. Zorbas. The control mechanism for the Myrias parallel computer system. *Computer Architecture News*, 16(4):21–30, Sept. 1988.

[8] T. Bergan et al. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *15th ASPLOS*, Mar. 2010.

[9] T. Bergan et al. Deterministic process groups in dOS. In *9th OSDI*, Oct. 2010.

[10] E. D. Berger et al. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, Oct. 2009.

[11] C. Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. In *17th PACT*, October 2008.

[12] R. L. Bocchino et al. Parallel programming must be deterministic by default. In *HotPar*. Mar. 2009.

[13] R. L. Bocchino et al. A type and effect system for deterministic parallel Java. In *OOPSLA*, Oct. 2009.

[14] S. Boyd-Wickizer et al. Corey: An operating system for many cores. In *8th OSDI*, Dec. 2008.

[15] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault-tolerance. *TOCS*, 14(1):80–107, Feb. 1996.

[16] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *FSE*, Aug. 2009.

[17] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *13th SOSP*, Oct. 1991.

[18] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *3rd OSDI*, pages 173–186, Feb. 1999.

[19] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59. 1998.

[20] H. Cui, J. Wu, and J. Yang. Stable deterministic multithreading through schedule memoization. In *9th OSDI*, Oct. 2010.

[21] J. Devietti et al. DMP: Deterministic shared memory multiprocessing. In *14th ASPLOS*, Mar. 2009.

[22] G. W. Dunlap et al. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th OSDI*, Dec. 2002.

[23] G. W. Dunlap et al. Execution replay for multiprocessor virtual machines. In *VEE*, Mar. 2008.

[24] S. A. Edwards, N. Vasudevan, and O. Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *DATE*, Mar. 2008.

[25] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *19th SOSP*, Oct. 2003.

[26] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *15th SOSP*, Dec. 1995.

[27] B. Ford et al. Microkernels meet recursive virtual machines. In *2nd OSDI*, pages 137–151, 1996.

[28] K. Gharachorloo et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th ISCA*, pages 15–26, May 1990.

[29] git: the fast version control system. `http://git-scm.com/`.

[30] A. Haeberlen et al. Accountable virtual machines. In *9th OSDI*, Oct. 2010.

[31] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *21st SOSP*, Oct. 2007.

[32] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *TOPLAS*, 7(4):501–538, Oct. 1985.

[33] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th ISCA*, pages 289–300, May 1993.

[34] A. Joshi et al. Detecting past and present intrusions through vulnerability-specific predicates. In *20th SOSP*, pages 91–104. 2005.

[35] F. Kaashoek et al. 6.828: Operating system engineering. `http://pdos.csail.mit.edu/6.828/`.

[36] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, pages 471–475. 1974.

[37] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX*, pages 1–15, Apr. 2005.

[38] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, Sept. 1979.

[39] T. J. Leblanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, Apr. 1987.

[40] E. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[41] J. Liedtke. On micro-kernel construction. In *15th SOSP*, 1995.

[42] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *13th ASPLOS*, pages 329–339, Mar. 2008.

[43] M. Musuvathi et al. Finding and reproducing Heisenbugs in concurrent programs. In *8th OSDI*. 2008.

[44] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *14th ASPLOS*, Mar. 2009.

[45] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[46] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel programs. In *PADD '88*, pages 124–129. 1988.

[47] D. S. Parker, Jr. et al. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3), May 1983.

[48] D. E. Porter et al. Operating system transactions. In *22nd SOSP*, Oct. 2009.

[49] C. Sadowski, S. N. Freund, and C. Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. In *18th ESOP*, Mar. 2009.

[50] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(4):299–319, Dec. 1990.

[51] J. T. Schwartz. The burroughs FMP machine, Jan. 1980. Ultracomputer Note #5.

[52] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, Feb. 1997.

[53] D. B. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th SOSP*, 1995.

[54] R. von Behren et al. Capriccio: Scalable threads for internet services. In *SOSP'03*.

[55] B. Walker et al. The LOCUS distributed operating system. *OSR*, 17(5), Oct. 1983.

[56] L. Wittie. The Bugnet distributed debugging system. In *Making Distributed Systems Work*, Sept. 1986.

[57] S. C. Woo et al. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd ISCA*, pages 24–36, June 1995.

Stable Deterministic Multithreading through Schedule Memoization

Heming Cui, Jingyue Wu, Chia-che Tsai, Junfeng Yang
{*heming, jingyue, ct2459, junfeng*}*@cs.columbia.edu*
Computer Science Department
Columbia University
New York, NY 10027

Abstract

A deterministic multithreading (DMT) system eliminates nondeterminism in thread scheduling, simplifying the development of multithreaded programs. However, existing DMT systems are unstable; they may force a program to (ad)venture into vastly different schedules even for slightly different inputs or execution environments, defeating many benefits of determinism. Moreover, few existing DMT systems work with server programs whose inputs arrive continuously and nondeterministically.

TERN is a *stable* DMT system. The key novelty in TERN is the idea of *schedule memoization* that memoizes past working schedules and reuses them on future inputs, making program behaviors stable across different inputs. A second novelty in TERN is the idea of *windowing* that extends schedule memoization to server programs by splitting continuous request streams into windows of requests. Our TERN implementation runs on Linux. It operates as user-space schedulers, requiring no changes to the OS and only a few lines of changes to the application programs. We evaluated TERN on a diverse set of 14 programs (*e.g.*, Apache and MySQL) with real and synthetic workloads. Our results show that TERN is easy to use, makes programs more deterministic and stable, and has reasonable overhead.

1 Introduction

Multithreaded programs are difficult to write, test, and debug. A key reason is nondeterminism: different runs of a multithreaded program may show different behaviors, depending on how the threads interleave [35].

Two main factors make threads interleave nondeterministically. The first is *scheduling*, how the OS and hardware schedule threads. Scheduling nondeterminism is not essential and can be eliminated without affecting correctness for most programs. The second is *input*, what data (*input data*) arrives at what time (*input timing*). Input nondeterminism sometimes is essential because major changes in inputs require different schedules. However, frequently input nondeterminism is not essential and the same schedule can be used to process many different inputs (§2.2). We believe nonessential nondeterminism should be eliminated in favor of determinism.

Deterministic multithreading (DMT) systems [13, 22, 41] make threads more deterministic by eliminating scheduling nondeterminism. Specifically, they constrain a multithreaded program such that it always uses the same thread schedule for the same input. By doing so, these systems make program behaviors repeatable, increase testing confidence, and ease bug reproduction.

Unfortunately, though existing DMT systems eliminate scheduling nondeterminism, they do not reduce input nondeterminism. In fact, they may aggravate the effects of input nondeterminism because of their design limitation: when scheduling the threads to process an input, they consider only this input and ignore previous similar inputs. This stateless design makes schedules over-dependent on inputs, so that a slight change to inputs may force a program to (ad)venture into a vastly different, potentially buggy schedule, defeating many benefits of determinism. We call this the *instability* problem. This problem is confirmed by our results (§8.2.1) from an existing DMT system [13].

In fact, even with the same input, existing DMT systems may still force a program into different schedules for minor changes in the execution environment such as processor type and shared library. Thus, developers may no longer be able to reproduce bugs by running their program on the bug-inducing input, because their machine may differ from the machine where the bug occurred.

This paper presents TERN, a schedule-centric, stateful DMT system. It addresses the instability problem using an idea called *schedule memoization* that memoizes past working schedules and reuses them for future inputs. Specifically, TERN maintains a cache of past schedules and the input constraints required to reuse these schedules. When an input arrives, TERN checks the input against the memoized constraints for a compatible sched-

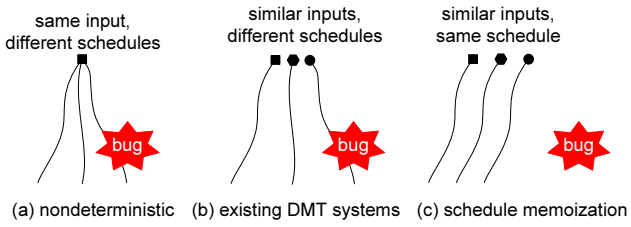


Figure 1: *Advantage of schedule memoization.* Each solid shape represents an input, and each curved line a schedule. Schedule memoization reuses schedules when possible, avoiding bugs in unknown schedules and making program behaviors repeatable across similar inputs.

ule. If it finds one, it simply runs the program while enforcing this schedule. Otherwise, it runs the program to memoize a schedule and the input constraints of this schedule for future reuse. By reusing schedules, TERN avoids potential errors in unknown schedules. This advantage is illustrated in Figure 1.

A real-world analogy to schedule memoization is the natural tendencies in humans and animals to follow familiar routes to avoid possible hazards along unknown routes. Migrant birds, for example, often migrate along fixed "flyways." We thus name our system after the Arctic Tern, a bird species that migrates the farthest among all migrants [2].

A second advantage of schedule memoization is that it makes schedules explicit, providing flexibility in deciding when to memoize certain schedules. For instance, TERN allows developers to populate a schedule cache offline, to avoid the overhead of doing so online. Moreover, TERN can check for errors (*e.g.*, races) in schedules and memoize only the correct ones, thus avoiding the buggy schedules and amortizing the cost of checking for errors.

To make TERN practical, it must handle server programs which frequently use threads for performance. These programs present two challenges for TERN: (1) they often process client inputs (requests) as they arrive, thus suffering from input timing nondeterminism, which existing DMT systems do not handle and (2) they may run continuously, making their schedules effectively infinite and too specific to reuse.

TERN addresses these challenges using a simple idea called *windowing*. Our insight is that server programs tend to return to the same quiescent states. Thus, TERN splits the continuous request stream of a server into *windows* and lets the server quiesce in between, so that TERN can memoize and reuse schedules across windows. Within a window, it admits requests only at fixed schedule points, reducing timing nondeterminism.

We implemented TERN in Linux. It runs as "parasitic" user-space schedulers within the application's address space, overseeing the decisions of the OS scheduler and synchronization library. It memoizes and reuses synchronization orders as schedules to increase performance and reuse rates. It tracks input constraints using KLEE [17], a symbolic execution engine. Our implementation is software-only, works with general C/C++ programs using threads, and requires no kernel modifications and only a few lines of modification to applications, thus simplifying deployment.

We evaluated TERN on a diverse set of 14 programs, including two server programs Apache [10] and MySQL [4], a parallel compression utility PBZip2 [5], and 11 scientific programs in SPLASH2 [6]. Our workload included a Columbia CS web trace and benchmarks used by Apache and MySQL developers. Our results show that

1. TERN is easy to use. For most programs, we modified only a few lines to adapt them to TERN.
2. TERN enforces stability across different inputs. In particular, it reused 100 schedules to process 90.3% of a 4-day Columbia CS web trace. Moreover, while an existing DMT system [13] made three bugs inconsistently occur or disappear depending on minor input changes, TERN always avoided these bugs.
3. TERN has reasonable overhead. For nine out of fourteen evaluated programs, TERN has negligible overhead or improves performance; for the other programs, TERN has up to 39.1% overhead.
4. TERN makes threads deterministic. For twelve out of fourteen evaluated programs, the schedules TERN memoized can be deterministically reused barring the assumption discussed in §7.

Our main conceptual contributions are that we identified the instability problem in existing DMT systems and proposed two ideas, schedule memoization and windowing, to mitigate input nondeterminism. Our engineering contributions include the TERN system and its evaluation of real programs. To the best of our knowledge, TERN is the first stable DMT system, the first to mitigate input timing nondeterminism, and the first shown to work on programs as large, complex, and nondeterministic as Apache and MySQL. TERN demonstrates that DMT has the potential to be deployed today.

This paper is organized as follows. We first present a background (§2) and an overview of TERN (§3). We then describe TERN's interface (§4), schedule memoization for batch programs (§5), and windowing to extend TERN to server programs (§6). We then present refinements we made to optimize TERN (§7). Lastly, we show our experimental results (§8), discuss related work (§9), and conclude (§10).

2 Background

This section presents a background of TERN. We explain the instability problem of existing DMT systems (§2.1),

our choice of schedule representation in TERN (§2.2), and why we can reuse schedules across inputs (§2.3).

2.1 The Instability Problem

A DMT system is, conceptually, a function that maps an input I to a schedule S. The properties of this function are that the same I should map to the same S and that S is a feasible schedule for processing I. A stable DMT system such as TERN has an additional property: it maps similar inputs to the same schedule. Existing DMT systems, however, tend to map similar inputs to different schedules, thus suffering from the instability problem.

We argue that this problem is inherent in existing DMT systems because they are stateless. They must provide the same schedule for an input across different runs, using information only from the current run. To force threads to communicate (*e.g.*, acquire locks or access shared memory) deterministically, existing DMT systems cannot rely on physical clocks. Instead, they maintain a logical clock per thread that ticks deterministically based on the code this thread has run. Moreover, threads may communicate only when their logical clocks have deterministic values (*e.g.*, smallest across the logical clocks of all threads [41]). By induction, logical clocks make threads deterministic.

However, the problem with logical clocks is that for efficiency, they must tick at roughly the same rate to prevent a thread with a slower clock from starving others. Thus, existing DMT systems have to tie their logical clocks to low-level instructions executed (*e.g.*, completed loads [41]). Consequently, a small change to the input or execution environment may alter a few instructions executed, in turn altering the logical clocks and subsequent thread communications. That is, a small change to the input or execution environment may cascade into a much different (*e.g.*, correct vs. buggy) schedule.

2.2 Schedule Representation and Determinism

Previous DMT systems have considered two types of schedules: (1) a deterministic order of shared memory accesses [13, 22] and (2) a synchronization order (*i.e.*, a total order of synchronization operations) [41]. The first type of schedules are truly deterministic even if there are races, but they are costly to enforce on commodity hardware (*e.g.*, up to 10 times overhead [13]). The second type can be efficiently enforced (*e.g.*, 16% overhead [41]) because most code is not synchronization code and can run in parallel; however, they are deterministic only for inputs that lead to race-free runs [41, 46].

TERN represents schedules as synchronization orders for efficiency. An additional benefit is that synchronization orders can be reused more frequently than memory access orders (cf next subsection). Moreover, researchers have found that many concurrency errors are not data races, but atomicity and order violations [39]. These errors can be deterministically reproduced or avoided using only synchronization orders.

Program	Input Constraints for Schedule Reuse
PBZip2	Same number of file blocks (`NumBlocks` or `-b`) and threads (`-p`)
Apache	For groups of typical HTTP GET requests, same cache status and response sizes
fft	Same number of threads (`-p`)
lu	Same number of threads (`-p`), size of the matrix (`-n`), and block size (`-b`)
barnes	Same number of threads (`NPROC`) and values of variables `dtime` and `tstop`

Table 1: *Input constraints of five programs to reuse schedules.* Identifiers without a dash are configuration variables, and those with a dash are command line options.

Although data races may still make runs which reuse schedules nondeterministic, TERN is less prone to this problem than existing DMT systems [41] because it has the flexibility to select schedules. If it detects a race in a memoized schedule, it can simply discard this schedule and memoize another. This selection task is often easy because most schedules are race-free. In rare cases, TERN may be unable to find a race-free schedule, resulting in nondeterministic runs. However, we argue that input nondeterminism cannot be fully eliminated anyway, so we may as well tolerate some scheduling nondeterminism, following the end-to-end argument.

2.3 Why Can We Reuse Schedules?

This subsection presents an intuitive and an empirical argument to support our insight that we can frequently reuse schedules for many programs/workloads. Intuitively, synchronization operations map to developer intents of inter-thread control flow. By enforcing the same synchronization order, we fix the same inter-thread "path," but still allow many different inputs to flow down this path. (This observation is similarly made for sequential paths [11, 12, 26].)

To empirically validate our insight, we studied the input constraints to reuse schedules for five programs, including a parallel compression utility PBZip2; the Apache web server; and three scientific programs fft, lu, and barnes in SPLASH2. Table 1 shows the results for all programs studied. We found that the input constraints were often general, allowing frequent reuses of schedules. For instance, PBZip2 can use the same schedule to compress many different files, as long as the number of threads and the number of file blocks remain the same.

3 Overview

Our design of TERN adheres to the following goals:

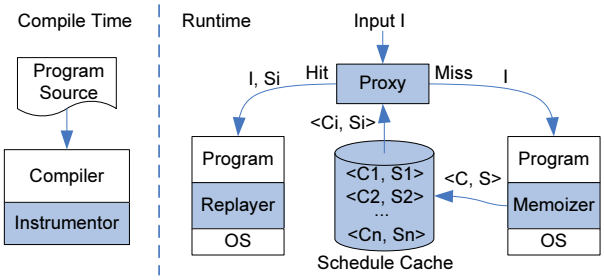


Figure 2: TERN *architecture.* Its components are shaded.

1. **Backward compatibility**. We design TERN for general multithreaded programs because of their dominance in parallel programs today and likely tomorrow. We design TERN to run in user-space and on commodity hardware to ease deployment.
2. **Stability**. We design TERN to bias multithreaded programs toward repeating their past, familiar schedules, instead of venturing into unfamiliar ones.
3. **Efficiency**. We design TERN to be efficient because it operates during the normal executions of programs, not replayed executions.
4. **Best-effort determinism**. We design TERN to make threads deterministic, but we sacrifice determinism when it contradicts the preceding goals.

The remaining of this section presents TERN's architecture (§3.1), workflow (§3.2), deployment scenarios (§3.3), and limitations (§3.4).

3.1 Architecture

Figure 2 shows the architecture of TERN and its five components: *instrumentor*, *schedule cache*, *proxy*, *replayer*, and *memoizer*. To use TERN, developers first annotates their application by marking the input data that may affect synchronization operations. They then compile their program with the *instrumentor*, which intercepts standard synchronization operations such as `pthread_mutex_lock()` so that at runtime TERN can control these operations. (We describe additional annotations and instrumentations that TERN needs in §4). The instrumentor runs as a plugin to LLVM [3], requiring no modifications to the compiler.

The *schedule cache* stores all memoized schedules and their input constraints. This cache can be marshalled to disk and read back upon program start, so that it need not be repopulated. Each memoized schedule is conceptually a tuple $\langle C, S\rangle$, where S is a synchronization order and C is the set of input constraints required to reuse S. (We explain the actual representation in §5.2).

At runtime, once an input I arrives, the *proxy* intercepts the input and queries the schedule cache for a constraint-schedule tuple $\langle C_i, S_i\rangle$ such that I satisfies C_i. On a cache hit, the proxy lets the *replayer* run the program on input I and enforce schedule S_i. On a cache miss, it lets the *memoizer* run the program on input I to memoize a new schedule.

```
1 :  main(int argc, char *argv[]) {
2 :    int i, nthread = argv[1], nblock = argv[2];
3 :    symbolic(&nthread, sizeof(int)); // mark input data
4 :    symbolic(&nblock, sizeof(int)); // that affects schedules
5 :    for(i=0; i<nthread; ++i)
6 :      pthread_create(worker); // create worker threads
7 :    for(i=0; i<nblock; ++i) {
8 :      block = read_block(i); // read i'th file block
9 :      worklist.add(block); // add block to work list
10:    }
11:  }
12:  worker() { // worker threads for compressing file blocks
13:    for(;;) {
14:      block = worklist.get(); // get a file block from work list
15:      compress(block);
16:    }
17:  }
```

Figure 3: Simplified PBZip2 code.

During a memoization run, the memoizer records all synchronization operations into a schedule S. It also computes C, the input constraints for reusing S, via symbolic execution [17]. The basic idea of symbolic execution is to track the outcomes of branches that observe symbolic data, in our case, the data marked by developers as affecting synchronizations. Once the memoization run ends, the set of branch outcomes we collected describes the input constraints needed to reuse the memoized schedule.

For determinism, the memoizer can optionally check a memoization run for data races. If it detects no races, it simply stores $\langle C, S\rangle$ into the schedule cache. Otherwise, it can discard the memoized schedule and rerun the program with a different scheduling algorithm to memoize another schedule.

The proxy performs an additional task for server programs to reduce input timing nondeterminism and to reuse schedules for these programs. Specifically, it buffers the requests of a server into a window with a fixed size. When the window becomes full, or remains partial for a predefined timeout, TERN runs the server to process the window as if the server were a batch program. It then lets the server quiesce before moving to the next window to avoid interference between windows.

3.2 Workflow and An Example

We illustrate how TERN works using PBZip2 as an example. Figure 3 shows the simplified code of PBZip2. Variables `nthread` and `nblock` affect synchronizations, so developers mark them by calling the TERN-provided method `symbolic()` (line 3 and line 4). This code spawns `nthread` worker threads, splits the file

```
// main              worker 1            worker 2
9: worklist.add();
                  14: worklist.get();
9: worklist.add();
                                      14: worklist.get();
```

Figure 4: Synchronization order of a PBZip2 run.

```
5: 0 < nthread ? true
5: 1 < nthread ? true
5: 2 < nthread ? false
7: 0 < nblock  ? true
7: 1 < nblock  ? true
7: 2 < nblock  ? false
```

Figure 5: Input constraints of a PBZip2 run.

into `nblock` blocks, and compresses them in parallel by calling `compress()`. To coordinate the worker threads, it uses a synchronized work list. (Note TERN tracks low-level synchronizations such as pthread primitives; we use a work list here only for clarity.)

Suppose we run PBZip2 with two threads on a two-block file. Suppose the schedule cache is empty and TERN runs the memoizer to memoize a new schedule. As PBZip2 runs, TERN controls and records the synchronization operations (line 9 and line 14). It also tracks the outcomes of branch statements that observe symbolic data (line 5 and line 7). At the end of the run, TERN records a schedule as shown in Figure 4. It also collects constraints as shown in Figure 5, which simplify to $nthread = 2 \wedge nblock = 2$.[1] It stores the schedule and the input constraints into the schedule cache.

If we run PBZip2 again with two threads on a different two-block file, TERN will check if variable `nthread` and `nblock` satisfy any set of constraints in the schedule cache. In this case, TERN will succeed. It will then reuse the schedule (Figure 4) to compress the file, even though the file data may differ completely.

3.3 Deployment Scenarios

We anticipate three ways users may deploy TERN to make their programs stable and deterministic.

Schedule-carrying code. Developers pre-populate a cache of correct, representative schedules on typical workloads, then ship their program with the cache hard-wired and marked read-only.

Online memoization. Users can turn on memoization at their local sites so that TERN can memoize schedules as the programs run on real inputs.

Shadow memoization. Since tracking input constraints is slow, users can configure TERN to memoize schedules asynchronously. Specifically, for an input that misses the schedule cache, the proxy runs the program as is, while forwarding a copy of the input to the memoizer.

Each deployment mode has pros and cons. The first mode makes a program stable and deterministic across different sites, but may react poorly to site-specific workloads. The second mode updates the schedule cache based on site-specific workloads, but may be slow because memoization runs tend to be slow. The last approach avoids the slowdown, but allows a program to run nondeterministically when an input misses the schedule cache. For server programs with high performance requirements, we recommend the first and the third modes.

3.4 Limitations

Determinism. TERN aims for best-effort determinism for reasons discussed in §2.2. If TERN is unable to find a race-free schedule for an input, the run may be nondeterministic. We foresee several strategies to handle this corner case while adhering to the other goals of TERN. For instance, we can instrument the program to fix the detected races or apply one of the existing DMT algorithms to resolve the races deterministically. The advantage of combining these techniques with TERN is that we apply these expensive techniques only to a small portion of schedules, and use TERN to efficiently handle the common case. We leave these ideas for future work.

Applicability. We anticipate our approach will work well for many programs/workloads as long as (1) they can benefit from determinism and stability, (2) their constraints can be tracked by TERN, (3) their schedules can be frequently reused, and (4) if windowing is needed, their inputs can be buffered. For programs/workloads that violate these assumptions, TERN may work poorly. These programs/workloads may include parallel simulators that require nondeterminism for statistical results, GUI programs that cannot buffer user actions for latency reasons, randomly generated workloads that prevent schedule reuses, and programs whose schedules depend on floating point inputs (which cannot be tracked by TERN's underlying symbolic execution engine).

Manual annotation. TERN requires manual annotations. However, this annotation overhead tends to be small. (See §7.4 for how TERN reduces this overhead and §8.1 for an evaluation of this overhead). This overhead may be further reduced using simple static analysis.

4 Interface

Table 2 shows TERN's annotation interface which developers and the instrumentor use to annotate multi-threaded programs. The annotations fall into four categories: (1) `symbolic()` for marking data that may affect schedules; (2) task boundary annotations for marking the beginning and end of logical tasks, in case threads get reused for different logical tasks (§6); (3) wrap-

[1] Although in this example the constraints are collected from one thread, TERN can actually collect constraints from multiple threads.

Annotations	Inserted by	Semantics
symbolic(*data*, *len*)	Developer	Marks data that may affect schedules. The memoizer tracks constraints on this data. The replayer checks this data against the memoized constraints.
begin_task() end_task()	Developer	Mark the beginning and end of a logical task. Often used to divide the executions of threads in a pool into separate tasks (§6).
lock_wrapper(*l*) unlock_wrapper(*l*)	Developer or TERN	Synchronization wrappers. The memoizer intercepts these operations for memoizing schedules, and the replayer intercepts them for reusing schedules.
before_blocking() after_blocking()	TERN	Inserted before and after blocking system calls. The memoizer logs the order of these calls. The replayer opportunistically enforces the same order of these calls.

Table 2: TERN *interface.* Some annotations are inserted by developers, and others are inserted by the instrumentor, indicated by Column **Inserted By**. Both the memoizer and the replayer use this interface, but they implement this interface differently (§5).

pers to synchronization operations (more examples in the next paragraph); and (4) hook functions inserted around blocking system calls, which TERN memoizes because blocking systems calls are natural scheduling points.

Currently TERN hooks 28 pthread operations (*e.g.*, `pthread_mutex_lock()`, `pthread_create()`, and `pthread_cond_wait()`). It also handles common atomic operations such as `atomic_dec()` and `atomic_inc()`. It hooks eight blocking system calls (*e.g.*, `sleep()`, `accept()`, `recv()`, `select()`, and `read()`). These hooks are sufficient to run the programs evaluated, and we can easily add more.

Developers manually insert annotations in the first two categories. They also annotate custom synchronizations (*e.g.*, custom spin locks). TERN's instrumentor automatically hooks standard synchronization and blocking system calls. These annotations allow TERN's memoizer and replayer to run as "parasitic" user-space schedulers that oversee the scheduling decisions of the OS and synchronization library, requiring no modifications to either.

5 Schedule Memoization

This section presents the idea of schedule memoization in the context of batch programs. We describe how TERN memoizes schedules (§5.1), tracks input constraints (§5.2), merges a schedule into the schedule cache (§5.3), and reuses schedules (§5.4).

5.1 Memoizing Schedules

To memoize schedules, the memoizer controls and logs synchronization operations. By default, it uses a simple round-robin (RR) algorithm that forces each thread to do synchronizations in turn. One advantage of this algorithm is that independent sites may memoize the same schedules, making program behaviors deterministic and stable across sites.

The memoizer implements this algorithm by implementing the wrapper functions in Table 2. Figure 6 shows the wrappers to `pthread_mutex_lock()` and `pthread_mutex_unlock()`. The memoizer maintains a queue of active threads. Only the thread at the head of the queue "has the turn" (line 4 and 14). Once

```
1 :  queue_t activeq, waitq[N];
2 :  pthread_mutex_lock_wrapper(pthread_mutex_t *mutex) {
3 :   retry:
4 :    while(self()!=activeq.head); // wait for our turn
5 :    if(!phtread_mutex_trylock(mutex)) { // mutex acquired
6 :      append(schedule, self()); // add tid to schedule
7 :      move(self(), activeq.tail); // give turn to next thread
8 :      return;
9 :     }
10:     move(self(), waitq[mutex].tail); // deterministically wait
11:     goto retry; // wait for our turn again
12: }
13: pthread_mutex_unlock_wrapper(pthread_mutex_t *mutex) {
14:     while(self()!=activeq.head); // wait for our turn
15:     pthread_mutex_unlock(mutex); // mutex released
16:     wake_up(waitq[mutex].head); // deterministically wake up
17:     append(schedule, self()); // add tid to schedule
18:     move(self(), activeq.tail); // give turn to next thread
19: }
```

Figure 6: *The memoizer's round-robin scheduling algorithm.*

the thread is done with the operation, it gives up the turn by moving itself to the tail of the queue (line 7 and 18).

We explain three subtleties of the code. First, to avoid the deadlock scenario when the head of the queue attempts to grab an unavailable mutex, we call the non-blocking lock operation instead of the blocking one (line 5). If the mutex is not available, the thread gives up its turn and waits on a TERN-maintained wait queue (line 10). TERN uses its own wait queues to avoid nondeterministic wakeup orders in pthread library. Second, we log synchronizations (line 6 and line 17) only when the thread has the turn, so that the log faithfully reflects the actual order of synchronizations. Lastly, we maintain our internal thread IDs to avoid nondeterminism in the OS thread IDs across runs. Function `self()` returns this internal ID for the current thread (line 6 and line 17).

The memoizer allows a thread to break out of the round-robin when the thread has waited for its turn for over a second. The rationale is that if a thread has waited too long, the current schedule will likely perform poorly in reuse runs. However, such timeouts do not affect nondeterminism, because the memoizer still logs the order of

the occurred operations and the replayer simply enforces the same order. In our experiments, we never observed such timeouts because most threads synchronize or call blocking system calls frequently.

Unlike previous DMT systems, TERN has the flexibility to select scheduling algorithms. In addition to the RR algorithm, it implements a first-come first-served (FCFS) algorithm that lets threads run as is. If the memoizer detects a race using RR, it can restart the run and switch to FCFS. Implementing FCFS requires only minor modifications to the algorithm presented in Figure 6. Specifically, we replace line 4 and line 14 with a lock operation; line 7, line 10, and line 18 with an unlock operation; and line 16 a NOP.

In addition to synchronizations, the memoizer includes the hooks around blocking system calls (§4) in the schedule it memoizes because blocking system calls are natural scheduling points. However, the replayer will only opportunistically replay these hooks when reusing a schedule because the returns from blocking system calls are driven by the program's environment.

5.2 Tracking Input Constraints

Given the symbolic data marked by developers, the memoizer tracks the constraints on this data by tracking (1) what data is derived from the symbolic data and (2) the outcomes of the branch statements that observe this symbolic and derived data. At the end of this memoization run, the set of branch outcomes together describe the constraints to place on the symbolic data required to reuse the memoized schedule. That is, if an input satisfies these constraints, we can re-run the program in the same way as the memoization run. The constraints collected this way may be over-constraining if developers annotate too much data as symbolic. We describe a technique to address this problem in §7.4.

TERN leverages KLEE [17], an open-source symbolic execution engine to track input constraints. To adapt KLEE to TERN, we made two key modifications. First, KLEE works only with sequential programs, thus we extended it to support threads. Specifically, we modified KLEE to spawn a new KLEE instance for each new thread. At the end of the run, we unify the constraints collected from each thread as the input constraints of the schedule. Second, we simplified KLEE to only collect constraints without solving them, because unlike KLEE, TERN need not explore different execution paths.

5.3 Merging Schedules into the Schedule Cache

Once TERN memoized a schedule S and its constraints C, TERN stores the tuple into the schedule cache. Although the schedule cache is conceptually a set of $\langle C, S\rangle$ tuples, its actual structure is a decision tree because a program may incrementally read inputs from its environment, calling `symbolic()` multiple times. For example, the code in Figure 3 calls `symbolic()` twice.

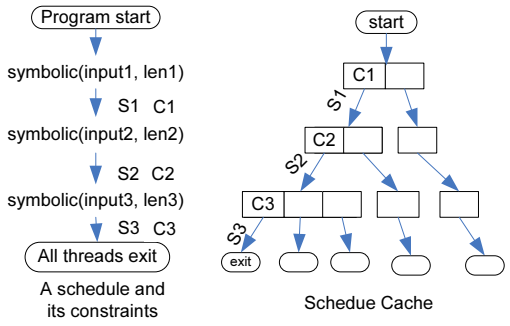


Figure 7: *Decision tree of* TERN*'s schedule cache.*

Figure 7 illustrates how TERN constructs the decision tree of the schedule cache. Given a $\langle C, S\rangle$ tuple, TERN breaks it down to sub-tuples $\langle C_i, S_i\rangle$ separated by `symbolic()` calls, where S_i contains the synchronization operations logged and C_i contains the constraints collected between the i^{th} and $(i+1)^{th}$ `symbolic()` calls. It then merges the sub-tuples into the i^{th} level of the decision tree.

TERN avoids merging redundant tuples into the cache. That is, if the cache contains a tuple with less restrictive constraints that the tuple being merged, TERN simply discards the new tuple. Note that the tuples may overlap (*i.e.*, one input satisfies more than one set of constraints), and TERN simply returns the first match if there are multiple matches.

To speed up cache lookup, TERN sorts all $\langle C_i, S_i\rangle$ tuples within the same decision node based on their *reuse rates*, defined as the number of successful reuses of S_i over the number of inputs that have satisfied C_i. Reusing a schedule may fail even if the input satisfies the schedule's input constraints (cf next subsection). However, by sorting the tuples based on reuse rates, we automatically prefer good schedules over bad ones that have many failed reuse attempts. To bound the size of the schedule cache, TERN can throw away bad schedules based on reuse rates. However, we have not found the need to do so because the schedule cache is often small.

5.4 Reusing Schedules

To reuse a schedule, TERN must check that the input satisfies the input constraints of the schedule. To do so, it maintains an iterator to the decision tree of the schedule cache. The iterator starts from the root. As the program runs and calls `symbolic()`, TERN moves the iterator down the tree. It checks if the data passed into a `symbolic()` call satisfies any set of constraints stored at the corresponding decision tree node and, if so, enforces the corresponding schedule.

```
1 :  pthread_mutex_lock_wrapper(mutex) {
2 :    down(sem[self()]); // wait for our turn
3 :    pthread_mutex_lock(mutex);
4 :    next = shift schedule; // find next thread in schedule
5 :    up(sem[next]); // wake up next thread
6 :  }
```

Figure 8: *Pseudo code of the replayer.*

The performance of the replayer is crucial because it runs during a program's normal executions. To efficiently enforce a synchronization order, the replayer uses a technique we call *semaphore relay*. Specifically, the replayer assigns each thread a semaphore. Before doing a synchronization operation, a thread has to wait on its semaphore for its turn. Once it is done with the operation, it passes the turn to the next thread in the schedule by signaling the semaphore of the next thread. Compared to an approach using locks or condition variables, semaphore relay avoids unnecessary lock contentions. Figure 8 illustrates semaphore relay using the replayer's `pthread_mutex_lock()` wrapper.

We note several subtleties of the pseudo code in Figure 8. First, we do not use non-blocking lock operations (line 3) as in Figure 6 because the memoizer only logs successful lock acquisitions. Second, the replayer maintains internal thread IDs the same way as the memoizer to avoid mismatches. Lastly, the `down()` (line 2) is actually a timed wait (with a default 0.1ms timeout), so that a thread can break out of a schedule when the dynamic load mismatches the schedule's assumptions. Note that these timeouts merely cause delays and do not affect correctness. They rarely occurred in our experiments.

6 Windowing

Server programs present two challenges for TERN. First, they are more exposed to timing nondeterminism than batch programs because their inputs (client requests) arrive nondeterministically. Second, they often run continuously, making their schedules too specific to reuse.

TERN addresses these challenges using a simple idea called *windowing*. Our insight is that server programs tend to return to the same quiescent states. Thus, instead of processing requests as they arrive, TERN breaks a continuous request stream down to windows of requests. Within each window, it admits requests only at fixed points in the current schedule. If no requests arrive at an admission point for a predefined timeout, TERN simply proceeds with the partial window. While a window is running, TERN buffers newly arrived requests so that they do not interfere with the running window. With this approach, TERN can memoize and reuse schedules across (possibly partial) windows. The cost of windowing is that it may reduce concurrency and degrade server throughput and speed. However, our experiments show that this cost is reasonable and justified by the gain in determinism and stability.

To buffer requests, TERN needs to know when a server receives a request and when it is done processing the request. Inferring these task boundaries based on thread creation and exit is unreliable because server programs frequently use thread pools. Thus, TERN currently lets developers annotate these boundaries using `begin_task()` and `end_task()`. Manually locating task boundaries is often easy: a request tends to begin after an `accept()` of a client connection and ends after the server sends out a reply.

Exposing hidden states. The assumption of windowing is that a server program returns to the same state when it quiesces. However, in practice, server states evolve over time. For instance, when Apache first serves a page, it may load the page from disk and cache it in memory. When this page is requested again, Apache can serve it directly from its cache.

These state changes may affect schedules. In the example above, Apache will perform different synchronizations for the two runs. Thus, for TERN to accurately select a schedule to reuse, it must know the hidden states that affect schedules. Currently TERN lets developers annotate such hidden states using `symbolic()`. Doing so is often straightforward. For instance, we inserted a `symbolic()` call to mark the return of Apache's `cache_find()` as symbolic.

Exposing hidden states may not always be easy. We thus created a technique to tolerate missed `symbolic()` annotations. The basic idea is to store backup schedules under the same set of input constraints to tolerate annotation inaccuracy. For instance, suppose a `symbolic()` had not been missed, TERN would have memoized two different constraint-schedule tuples $\langle C_1, S_1 \rangle$ and $\langle C_2, S_2 \rangle$. However, because of the missed annotation, TERN missed the corresponding constraints, wrongly collapsing C_1 and C_2 into the same set C. Now the two original tuples become $\langle C, S_1 \rangle$ and $\langle C, S_2 \rangle$, which appear redundant. Instead of discarding one of these seemingly redundant schedules, TERN will store both schedules with the same set of constraints. To select between these schedules, TERN can select the one with higher reuse rate, which likely matches the hidden state of the program.

7 Refinements

This section describes four refinements we made, one for determinism (§7.1) and three for speed (§7.2-§7.4).

7.1 Detecting Data Races

As discussed in §2.2, if a memoized schedule allows data races, runs reusing this schedule may become nondeterministic. Thus, for determinism, we would like to de-

```
// T1          // T2
++x;
lock(l1);
                lock(l2);
                ++x;
```

Figure 9: *A conventional race, not a schedule race.*

```
// T1          // T2
lock(l1);
a[i]++;         lock(l2);
                a[j]−−;
unlock(l1);
                unlock(l2);
```

Figure 10: *A symbolic race that occurs only when $i = j$.*

tect races in memoized schedules and discard them from the schedule cache. A general race detector would flag too many races for TERN because it detects conventional races with respect to the original synchronization constraints of the program, whereas we want to detect races with respect to the order constraints of a schedule [46] (call them *schedule races*). Figure 9 shows a conventional race, but not a schedule race because the synchronization order shown "kills" the race.

Thus, we built a simple race detector to detect schedule races. It runs with the memoizer and is happens-before based. It considers one memory access happens before another with respect to the synchronization order the memoizer records. Sometimes a pair of instructions may appear to be a race, when in fact their relative order does not alter a run. For instance, a write-write race is benign if both instructions write the same value. Similarly, a read-write race is benign if the value written by one instruction does not affect the value read by another. Our race detector prunes these benign races.

Our detector also flags *symbolic races*, the races that are data-dependent on inputs. Figure 10 shows an example. Both variables i and j are inputs, and the race occurs only when $i = j$. The risk of a symbolic races is that it may be absent in a memoization run and thus skip detection, but show up nondeterministically in a reuse run. To detect symbolic races, our race detector queries the underlying symbolic execution engine for pointer equality. For example, to detect the race in Figure 10, it would query the underlying symbolic execution engine for the satisfiability of $\&a[i] = \&a[j]$. It flags a symbolic race if this constraint is satisfiable. Once a symbolic race is flagged, TERN adds additional input constraints to ensure that the race does not occur in reuse runs. For Figure 10, we would add $\&a[i] \neq \&a[j]$, which simplifies to $i \neq j$.

Our race detector can detect all schedule races in a memoization run. It can also detect all symbolic races if developers correctly annotate all data that affect synchronization operations and memory locations accessed. If this assumption holds and our race detector reports no races in a memoization run, TERN ensures that the memoized schedule can be deterministically reused.

7.2 Skipping Unnecessary Synchronizations

When reusing a schedule, TERN enforces a total synchronization order according to the schedule. These TERN-enforced execution order constraints are more stringent than the constraints enforced by the original synchronizations in the program. Thus, for speed, TERN can actually skip these unnecessary synchronizations. In our current implementation, we skip `sleep()`, `usleep()`, and `pthread_barrier_wait()` because they are frequently used. We found that this optimization was quite effective and even made programs run faster than nondeterministic execution (§8.3).

7.3 Simplifying Constraints

To reuse a schedule, TERN must check if the current input satisfies the constraints of the schedule. The overhead of this check depends on the number of constraints, yet the set of constraints TERN collects may not always be in simplified form. That is, a subset of the constraints may imply the entire set. For example, consider a loop "`for(int i=0;i!=n;++i)`" with a symbolic bound n. When running this code with $n = 10$, we will collect a set of constraints $\{0 \neq n, 1 \neq n, ..., 10 = n\}$, but the last constraint alone implies the entire set.

To simplify constraints, TERN uses a greedy algorithm. Given a set of constraints C, it iterates through each constraint c, and checks if $C/\{c\}$ implies $\{c\}$. If so, it simply discards c. Our observation is that constraints collected later in a run tend to be more compact than the earlier ones. Thus, when pruning constraints, we start from the ones collected earlier. Although we could have used the underlying symbolic execution engine to simplify constraints, it lacks this domain knowledge and may perform poorly.

7.4 Slicing Out Irrelevant Branches

A branch statement may observe a piece of symbolic data but perform no synchronization operation in either branch. The constraints collected from this branch are unlikely to affect schedules. If we include irrelevant constraints in the input constraints of a schedule, we not only increase constraint checking time, but also preclude legal reuses of the schedule.

To address this problem, TERN employs a simple static analysis to automatically prune likely irrelevant constraints. At the heart of this technique is a slicing analysis that identifies branch statements unlikely to affect synchronization operations. Specifically, given a branch statement s, this analysis computes s_d, the immediate post-dominator [8] of s, and marks s as irrelevant if no synchronization operations are between s and s_d. Although simple, this technique reduced constraint checking time significantly (§8.3). However, we note that our analysis is unsound because it ignores data dependencies. Thus, we plan to implement a sound slicing algorithm [21] in our future work.

Program	Size	Symbolic	Task	Sync	Total
Apache	464K	4	2	0	6 (+1)
MySQL	1,182K	1	2	0	3 (+28)
PBZip2	1,551	3	N/A	0	3
fft	1,403	4	N/A	0	4
lu	1,265	3	N/A	0	3
barnes	1,954	9	N/A	0	9
radix	661	4	N/A	0	4
fmm	3,208	8	N/A	1	9
ocean	6,494	5	N/A	0	5
volrend	18,082	1	N/A	1	2
water-spatial	1,573	9	N/A	0	9
raytrace	5,808	3	N/A	0	3
water-nsquared	1,188	10	N/A	0	10
cholesky	3,683	3	N/A	1	4

Table 3: *Statistics of programs evaluated.* **Size** counts the lines of code for each program. **Symbolic** counts the symbolic variables we marked. **Task** counts the task boundary annotations (`begin_task()` and `end_task()`) we inserted. **Sync** counts the annotations for custom synchronizations we inserted. The numbers in parenthesis under **Total** count miscellaneous changes.

	Nondet			CoreDet			Tern		
-p2	✔	✔	✔	✔	✖	✔	✔	✔	✔
-p4	✔	✔	✔	✖	✖	✔	✔	✔	✔
-p8	✔	✔	✔	✖	✖	✖	✔	✔	✔
Args.	**-m10**	**12**	**14**	**-m10**	**12**	**14**	**-m10**	**12**	**14**

Table 4: *Bug stability results on SPLASH2* `fft`. The leftmost column and the bottommost row show the command line arguments. Option **-p** specifies the number of threads, and **-m** the amount of computation (matrix size). Symbol ✖ indicates that the bug occured, and ✔ the bug never occured.

8 Evaluation

Our Tern implementation consists of 8,934 lines of C++ code, including 827 lines for the instrumentor implemented as an LLVM pass; 5,451 lines for the proxy, schedule cache, memoizer, and replayer; and 2,656 lines for modifications to KLEE.

We evaluated Tern on a diverse set of 14 programs, ranging from two server programs, Apache and MySQL, to one parallel compression utility, PBZip2, to 11 scientific programs in SPLASH2.[2]

Our main evaluation machine is a 2.66 GHz quad-core Intel machine with 4 GB memory running Linux 2.6.24. When evaluating Tern on server programs, we ran the server on this machine and the client on another to avoid unnecessary contention. These machines are connected via 1Gbps LAN. We compiled all programs down to machine code using `llvm-gcc -O2` and LLVM's bitcode compiler `llc`.

We focused our evaluation on four key questions:

1. Is Tern easy to use (§8.1)?
2. Does Tern make multithreaded programs stable across different inputs (§8.2)?
3. Does Tern incur high overhead (§8.3)?
4. Does Tern make multithreaded programs deterministic on the same input (§8.4)?

8.1 Ease of Use

Table 3 summarizes the modifications we made to make the programs work with Tern. For each program but MySQL, we modified only 3-10 lines. For Apache, we marked the HTTP command, URL, HTTP version, and the return of `cache_find()` as symbolic (§6). For MySQL, we marked the SQL query. For PBZip2, we marked the number of threads and file blocks. (The number of file blocks is set in two places, contributing two symbolic annotations.) For all these scientific programs, we marked all input arguments as symbolic except those configuring output verbosity.[3] We marked three custom synchronization operations in three SPLASH2 programs. We made two miscellaneous changes to Apache and MySQL. The line counts are shown in parenthesis under the Total column. For Apache, we had to fix an uninitialized memory read in `ap_signal_server()` to make it work with KLEE. For MySQL, we wrote a 28-line function to mark the numbers in each SQL query as concrete (*i.e.*, not affecting schedules) to avoid making the input constraints too specific.

8.2 Stability

We evaluated Tern's stability via two sets of experiments. The first set compares it to an existing DMT system (§8.2.1), the second quantifies how frequently it can reuse schedules on real and synthetic workloads (§8.2.2).

8.2.1 Bug Stability

We compared Tern to CoreDet [13] in terms of *bug stability*: does a bug occur in one run but disappear in another when the input varies slightly? We ran three buggy SPLASH2 programs, fft, lu, and barnes, in three modes: nondeterministic execution (Nondet), with CoreDet, and with Tern. We varied their inputs by varying the number of threads and the amount of computation. For each program, execution mode, and input combination, we ran the program 100 times, and recorded whether the corresponding bug occurred.

We present only the fft results; the results of the other programs are similar. Table 4 shows the buggy behaviors of fft. In nondeterministic mode, the bug never occurred, despite that each run almost always yielded a new synchronization order. With CoreDet, slight changes

[2]The version of the SPLASH2 [36] we acquired has 12 programs, one of which does not compile on our evaluation machine.

[3]Note that we could have used a two-line loop to mark these arguments as symbolic. Instead, we report the total number of symbolic variables to avoid masking real data.

Program-Workload	Reuse Rates (%)	Schedules
Apache-CS	90.3%	100
SysBench-simple	94.0%	50
SysBench-tx	44.2%	109
PBZip2-usr	96.2%	90

Table 5: TERN *stability.* Column **Schedules** indicates the number of schedules in the schedule cache.

in computation made the bug occur or disappear. With TERN, the bug never occurred, and TERN reused only three schedules for all runs, one for each thread count.

8.2.2 Reuse Rates

We also quantified how frequently TERN could reuse schedules. Specifically, we measured the overall reuse rate, defined as the number of inputs processed using memoized schedules over the total number of inputs. The higher the reuse rates, the more stable the programs become. TERN had nearly 100% overall reuse rates for the scientific programs after a small number of memoization runs. Thus, we focused on Apache, MySQL, and PBZip2 in out experiments.

We used four workloads to evaluate overall reuse rates:

Apache-CS: a real 4-day trace from the Columbia CS website with 122,000 HTTP requests. We wrote a script to replay this trace at a rate of 100 concurrent requests per second.

SysBench-simple: SysBench [7] in simple mode. This synthetic workload consists of random select queries.

SysBench-tx: SysBench in transaction mode. This synthetic workload consists of random select, update, delete, and insert queries.

PBZip2-usr: a random selection of 10,000 files from `/usr` on our evaluation machine.

For each workload, we first randomly selected 1%-3% of the workload and ran the memoizer to populate the schedule cache. We then ran the entire workload with the replayer and measured the overall reuse rates. We ran eight worker threads for each program because they performed best (with or without TERN) with this setting.

Table 5 shows the results. For three out of the four workloads, TERN could reuse a small number of schedules to process over 90% of the inputs. For MySQL-tx, TERN had a lower overall reuse rate. The reasons are two fold. First, this workload makes it unlikely to reuse schedules because it mixes many randomly generated queries with different types and parameters. Second, we annotated only the SQL command as symbolic without exposing the hidden states of MySQL (§6) so that we could measure TERN's performance in an adversarial setting. Nonetheless, TERN managed to process 44.2% of inputs with a small number of schedules.

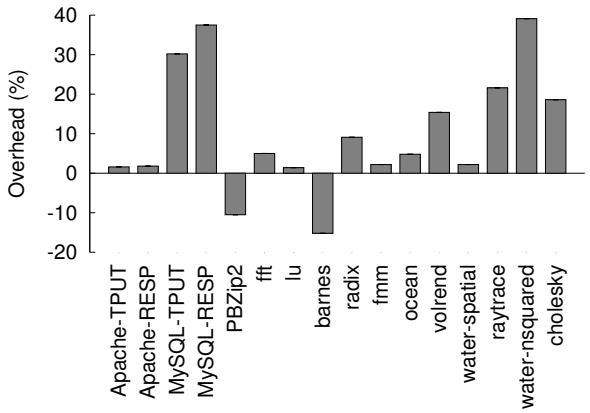


Figure 11: *Relative overhead of the replayer over nondeterministic execution.* Negative overhead means speedup.

8.3 Overhead

We used the following workloads to evaluate TERN's overhead. For Apache, we used ApacheBench [1] to repeatedly download a 50KB webpage. For MySQL, we used the SysBench-simple workload from the previous subsection. Both ApacheBench and SysBench are used by the server developers themselves. We made these benchmarks CPU bound by fitting the web or database in memory and by connecting the server and client via a 1 Gbps LAN. For PBZip2, we decompressed a 10 MB file. For SPLASH2 programs, we ran them typically for 10-100 ms. We measured the execution time for batch programs and the throughput (TPUT) and response time (RESP) for server programs. All numbers reported in this section were averaged over 50 runs.

The most performance-critical component is the replayer because it operates during the normal execution of a program. Figure 11 shows the relative overhead of the replayer over nondeterministic execution, the smaller the better. For seven out of the fourteen programs, the replayer performed almost identically to nondeterministic execution. For PBZip2 and barnes, TERN performed better. This speedup came partially from the optimization to remove unnecessary synchronizations, discussed in the next paragraph. TERN's overhead for MySQL, volrend, raytrace, water-nsquared, and choleskey is relatively large because these programs performed many synchronization operations over a short period of time. For instance, water-nsquared and cholesky both call `pthread_mutex_lock()` and `pthread_mutex_unlock()` in a tight loop.

We also measured the effects of skipping unnecessary synchronizations (§7.2). Figure 12 shows the results. This optimization significantly reduced the replayer's overhead for four programs. Specifically, it

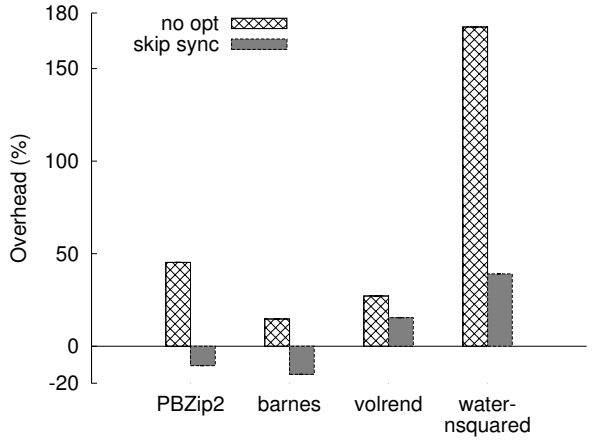


Figure 12: *Overhead reduction by skipping unnecessary synchronizations.* "no opt" indicates the baseline overhead.

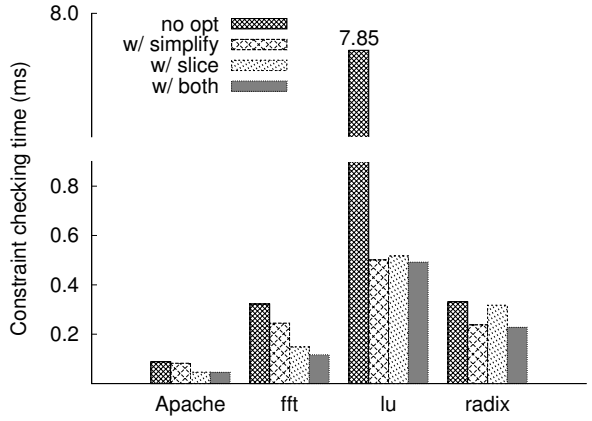


Figure 13: *Optimizations to speed up constraint checking.* Note the y-axis is broken. "no opt" indicates the baseline constraint checking time. "simplify" refers to the optimization in §7.3. "slice" refers to the optimization in §7.4.

made PBZip2 and barnes run faster than nondeterministic execution, and reduced the overhead of water-nsquared from 172.4% to 39.1%. Its effects on the other programs are negligible and thus not shown.

To reuse a schedule on an input, TERN must check the input against memoized constraints. Constraint checking can be costly, and TERN provides two optimizations to speed it up (§7.3 and §7.4). Figure 13 shows these optimizations can effectively speed up constraint checking for Apache, fft, lu, and radix. In particular, they reduced the constraint checking time for lu by 16x.

Compared to the replayer, the memoizer can run offline, thus its performance is not as critical. Table 6 shows that this slowdown can sometimes exceed 200x. The main reason is that KLEE, the symbolic engine used, interprets programs instead of running them natively. An instrumentation-based approach can greatly reduce this slowdown [16], which we plan to implement in our future work.

Program	Nondet	Memoization	Overhead (times)
Apache-TPUT	462.2 req/s	2.1 req/s	219.1
Apache-RESP	0.22 s	3.96 s	17.0
MySQL-TPUT	13779.3 req/s	172.2 req/s	79.0
MySQL-RESP	0.6 ms	61 ms	100.6
PBZip2	0.18 s	15.19 s	83.4

Table 6: *Overhead of the memoizer.*

Program	Error Description
Apache	Reference count decrement and check against 0 are not atomic.
PBZip2	Variable `fifo` is used in one thread after being freed by another.
fft	`initdonetime` and `finishtime` are read before assigned the correct values.
lu	Variable `rf` is read before assigned the correct value.
barnes	Variable `tracktime` is read before assigned the correct value.

Table 7: *Concurrency errors used in evaluation.*

8.4 Determinism

We evaluated TERN's determinism via three sets of experiments. The first set checked the memoized schedules for races (§8.4.1). The second evaluated TERN's ability to deterministically reproduce or avoid bugs (§8.4.2). The third measured how deterministic memory accesses are with and without TERN (§8.4.3).

8.4.1 Race Detection Results

When memoizing schedules for each of the 14 programs, we turned on TERN's race detector. We found that except for radix and cholesky, the schedules TERN memoized for all other programs were free of schedule races and symbolic races with respect to the symbolic data we annotated (§7.1). Our race detection result is not surprising because most schedules are indeed race free. It implies that, for runs that reuse the memoized schedules of all programs but radix and cholesky, TERN ensures determinism, barring the assumption discussed in §7.1.

8.4.2 Bug Determinism

We also evaluated how deterministically TERN could reproduce or avoid bugs. Table 7 lists five real concurrency bugs we used. We selected them because they were frequently used in previous studies [37, 39, 43, 44] and we could reproduce them on our evaluation machine. To measure bug determinism, we first memoized schedules for programs listed in Table 7. We then manually inserted `usleep()` to these programs to get alternate schedules.

Program	Length	Nondet	TERN	Ratio
Apache	148,058	86,215	10,821	7.97
PBZip2	1,234	161	69	2.33

Table 8: *Memory access determinism.* We traced memory accessed only from PBZip2, not the external BZip2 library.

We then ran the buggy programs again, reusing the memoized schedules. We also injected random delays into the reuse runs to perturb timing. We found that, TERN consistently reproduced or avoided all five bugs. We verified this result by inspecting the memoized schedules.

8.4.3 Memory Access Determinism

TERN enforces synchronization orders, which should make memory access orders more deterministic. We quantified this effect over Apache and PBZip2. Specifically, we instrumented Apache with LLVM to trace accesses to global variables and the heap, a crude approximation of shared memory. We ran Apache with TERN to serve five HTTP requests and collected a trace of memory accesses. We then repeated this experiment 20 times to collect 20 traces, and computed the average pairwise edit distance [52]. We then measured the same edit distance for Apache in nondeterministic execution mode and compared the two. We did the same comparison for PBZip2 with a decompression workload of 2MB. Table 8 shows the result. For Apache, runs with TERN were 7.97 times more deterministic than those without. For PBZip2, TERN was 2.33 times more deterministic, but the memory trace had only 1,234 accesses on average.

9 Related Work

Deterministic Execution TERN differs from existing DMT systems [13, 22, 41] by making threads stable, *i.e.*, repeating familiar behaviors across different inputs. Another difference is that TERN reduces timing nondeterminism for server programs through windowing.

The closest system to TERN in this category is Kendo [41], a software-only DMT system that also enforces synchronization orders instead of memory access orders for efficiency. COREDET [13] is another software-only DMT system that enforces deterministic memory access orders. Both systems are based on logical clocks and have been shown to work on scientific benchmarks, such as SPLASH2. The authors of COREDET have noted that a small modification to the original program leads to a much different COREDET-instrumented program, which the idea of schedule memoization may address. COREDET is a software implementation (with extensions) of DMP [22], a hardware DMT system .

Grace [14] proposes a novel approach to making C and C++ programs with fork-join parallelism behave like sequential programs. It runs each thread within a process and commits memory writes atomically and deterministically. It detects memory access conflicts efficiently using hardware page protection. Grace has been shown to perform and scale well on Phoenix benchmarks [45] and a Cilk [15] benchmark. Unlike Grace, TERN aims to make general multithreaded programs, not just fork-join programs, deterministic and stable.

Deterministic Replay Deterministic replay [9, 23, 24, 27, 31, 33, 34, 40, 44, 50, 51] aims to replay the exact recorded executions, whereas TERN "replays" memoized schedules on different inputs. Some recent deterministic replay systems include Scribe, which tracks page ownership to enforce deterministic memory access [34]; Capo, which defines a novel software-hardware interface and a set of abstractions for efficient replay [40]; PRES and ODR, which systematically search for a complete execution based on a partial one [9, 44]; and SMP-ReVirt, which uses clever page protection trick for recording the order of conflicting memory accesses [24].

Concurrency Errors The complexity in developing multithreaded programs has led to many concurrency errors [39]. A significant number of them are not data races, but atomicity and order errors [39], which can be deterministically reproduced or avoided using only synchronization orders.

Much work exists on concurrency error detection [25, 37, 38, 47, 55, 56], diagnosis [42, 43, 48], and correction [32, 53]. TERN aims to make the executions of multithreaded programs deterministic and stable, and is complementary to existing work on concurrency errors. Specifically, TERN can use existing work to detect and fix the errors in the schedules it selects. Moreover, even for programs free of concurrency errors, TERN still provides value by making their behaviors repeatable.

Symbolic Execution The combination of symbolic and concrete executions has been a hot research topic. Researchers have built scalable and effective symbolic execution systems to detect errors [16–18, 20, 28–30, 49, 54], block malicious inputs [21], and preserve privacy in error reports [19]. Compared to these systems, TERN applies symbolic execution to a new domain: tracking input constraints to reuse schedules.

10 Conclusion

We have presented TERN, the first DMT system that makes general multithreaded programs stable by repeating the same schedules on different inputs. TERN does so using schedule memoization: if a schedule is shown to work on an input, TERN memoizes the schedule; if a similar input arrives later, TERN simply reuses the memoized schedule. TERN is also the first DMT system to mitigate input timing nondeterminism for server programs.

Our TERN implementation runs on Linux. It requires

no new hardware, no modifications to the underlying OS or synchronization library, and only a few lines of modifications to the multithreaded programs. We evaluated TERN on a diverse set of real programs, including two server programs, one desktop program, and 11 scientific programs. Our results show that TERN is easy to use, makes programs more deterministic and stable, and has reasonable overhead. TERN is the first DMT system shown to work on applications as large, complex, and nondeterministic as MySQL and Apache. It demonstrates that DMT has the potential to be deployed today.

Acknowledgement

We thank Cristian Cadar, John Gallagher, Michael Kester, Emery Berger (our shepherd), and the anonymous reviewers for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We thank Shan Lu for providing some of the concurrency errors used in our evaluation. We thank Jane-Ellen Long for time management. Michael Kester wrote the script for replaying the HTTP trace from the Columbia CS website.

This work was supported by the National Science Foundation (NSF) through Contract CNS-1012633 and CNS-0905246 and the Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024 and FA8750-10-2-0253. Opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government.

References

[1] ab - Apache HTTP server benchmarking tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[2] Artici Terns - Wikipedia. `http://en.wikipedia.org/wiki/Arctic_Tern`.

[3] The LLVM Compiler Framework. `http://llvm.org`.

[4] MySQL Database. `http://www.mysql.com/`.

[5] Parallel BZIP2 (PBZIP2). `http://compression.ca/pbzip2/`.

[6] Stanford Parallel Applications for Shared Memory (SPLASH). `http://www-flash.stanford.edu/apps/SPLASH/`.

[7] SysBench: a system performance benchmark. `http://sysbench.sourceforge.net`.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.

[9] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 193–206, 2009.

[10] Apache Web Server. `http://www.apache.org`.

[11] T. Ball and J. R. Larus. Branch prediction for free. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 300–313, 1993.

[12] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, 1996.

[13] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10')*, pages 53–64, 2010.

[14] E. Berger, T. Yang, T. Liu, D. Krishnan, and A. Novark. Grace: Safe and efficient concurrent programming. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 2009*, 2009.

[15] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[16] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS '06)*, pages 322–335, Oct.–Nov. 2006.

[17] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 209–224, Dec. 2008.

[18] G. Candea, S. Bucur, and C. Zamfir. Automated software testing as a service. In *Proceedings of the 1st Symposium on Cloud Computing (SOCC '10)*, 2010.

[19] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 319–328, 2008.

[20] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective Symbolic Execution. In *Fifth Workshop on Hot Topics in System Dependability (HotDep '09)*, 2009.

[21] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 117–130, 2007.

[22] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 85–96, 2009.

[23] G. Dunlap, S. T. King, S. Cinar, M. Basrat, and P. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, Dec. 2002.

[24] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE '08)*, pages 121–130, 2008.

[25] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 237–252, Oct. 2003.

[26] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 85–95, 1992.

[27] D. Geels, G. Altekarz, P. Maniatis, T. Roscoey, and I. Stoicaz. Friday: Global comprehension for distributed replay. In *Proceed-*

ings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07), Apr. 2007.

[28] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 213–223, June 2005.

[29] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 206–215, 2008.

[30] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS '08: Proceedings of 15th Network and Distributed System Security Symposium*, Feb. 2008.

[31] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 193–208, Dec. 2008.

[32] H. Jula, D. Tralamazza, Z. Cristian, and C. George. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 295–308, Dec. 2008.

[33] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed Java applications. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS '00)*, pages 219–228, May 2000.

[34] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the 2010 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2010.

[35] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[36] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Proceedings of the first Workshop on the Evaluation of Software Defect Detection Tools (BUGS '05)*, June 2005.

[37] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Twelfth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '06)*, pages 37–48, Oct. 2006.

[38] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *SIGOPS Oper. Syst. Rev.*, 41(6):103–116, 2007.

[39] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339, 2008.

[40] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 73–84, 2009.

[41] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 97–108, 2009.

[42] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*, pages 135–145, Nov. 2008.

[43] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 25–36, Mar. 2009.

[44] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 177–192, 2009.

[45] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[46] M. Ronsse and K. De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17 (2):133–152, 1999.

[47] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, pages 391–411, Nov. 1997.

[48] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI '08)*, pages 11–21, June 2008.

[49] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, pages 263–272, Sept. 2005.

[50] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, pages 29–44, June 2004.

[51] VMWare Virtual Lab Automation. `http://www.vmware.com/solutions/vla/`.

[52] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[53] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 281–294, Dec. 2008.

[54] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (SP '06)*, pages 243–257, May 2006.

[55] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234, 2005.

[56] W. Zhang, C. Sun, and S. Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10')*, pages 179–192, 2010.

Enabling Configuration-Independent Automation by Non-Expert Users

Nate Kushman
Massachusetts Institute of Technology

Dina Katabi
Massachusetts Institute of Technology

Abstract

The Internet has allowed collaboration on an unprecedented scale. Wikipedia, Luis Von Ahn's ESP game, and reCAPTCHA have proven that tasks typically performed by expensive in-house or outsourced teams can instead be delegated to the mass of Internet computer users. These success stories show the opportunity for crowd-sourcing other tasks, such as allowing computer users to help each other answer questions like "How do I make my computer do X?". The current approach to crowd-sourcing IT tasks, however, limits users to text descriptions of task solutions, which is both ineffective and frustrating. We propose instead, to allow the mass of Internet users to help each other answer *how-to* computer questions by actually performing the task rather than documenting its solution.

This paper presents KarDo, a system that takes as input traces of low-level user actions that perform a task on individual computers, and produces an automated solution to the task that works on a wide variety of computer configurations. Our core contributions are machine learning and static analysis algorithms that infer state and action dependencies without requiring any modifications to the operating system or applications.

1 Introduction

Computer systems are becoming increasingly complex. As a result, users regularly encounter tasks that they do not know how to perform such as configuring their home router, removing a virus, or creating an email account. Many users do not have technical support, and hence their first, and often only, resort is a web search. Such searches, however, often lead to a disparate set of user forums written in ambiguous language. They rarely make clear which user configurations are covered by a particular solution; descriptions of different problems overlap; and many documents contain conjectured solutions that may not work. The net result is that users spend hours manually working through large collections of documents to try solutions that often fail to help them perform their task.

What a typical user really wants is a system that automatically performs the task for him, taking into account his machine configuration and global preferences, and asking the user only for information that cannot be automatically pulled from his computer. Today, however, automation requires experts to program scripts. This process is slow and expensive and hence unlikely to scale to the majority of tasks that users perform. For instance, a recent automation project at Microsoft succeeded in scripting only about 150 of the hundreds of thousands of knowledge-base articles in a period of 6 months [10].

This paper introduces KarDo, a system that enables the mass of Internet users to automate computer tasks. KarDo aims to build a database of automated solutions for computer tasks. The key characteristic of KarDo is that a user contributes to this database simply by performing the task. For lay users this means interacting with the graphical user interface, which manifests itself as a stream of windowing events (i.e., keypresses and mouse clicks). KarDo records the windowing events as the user performs the task. It then merges multiple such traces to produce a canonical solution for the task which encodes the various steps necessary to perform the task on different configurations and for different users. A user who comes across a task he does not know how to perform searches the KarDo database for a matching solution. The user can either use the solution as a tutorial that walks him through how to perform the task step by step, or ask KarDo to automatically perform the task for him.

The key challenge in automating computer tasks based on windowing events is that events recorded on one machine may not work on another machine with a different configuration. To address this problem, a system needs to understand the dependencies between the system state and the windowing events. While the system could track these dependencies explicitly by modifying

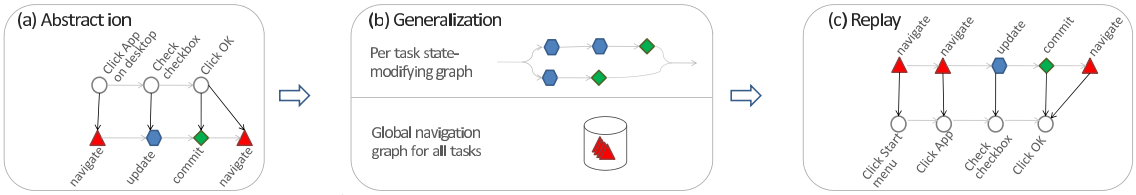


Figure 1: **Illustration of KarDo's three-stage design.**

the OS and potentially applications [18], such an approach presents a high deployment barrier and is hard to use for tasks that involve multiple machines (e.g., configuring a wireless router). KarDo therefore adopts an approach that implicitly infers system state dependencies, and does not require modifying the OS or applications. In particular, KarDo builds a model that maps windowing events to abstract actions that capture impact on system state: UPDATE and COMMIT actions, which actually modify system state, and NAVIGATE actions, which simply open or close windows but do not modify system state. KarDo performs this mapping automatically using machine learning. It then runs a set of static analysis algorithms on these sequences of abstract actions to produce a canonical solution which can perform the task on various different configurations. The system operates in 3 stages, described below and shown in Fig. 1.

(a) Abstraction. KarDo first captures the context around each windowing event (e.g, the associated application, window, widget *etc.*) using the accessibility interface, which was originally developed for visually impaired users and is supported by modern operating systems [8, 5]. KarDo then extracts from the context a per event feature vector, which it uses in a machine learning algorithm to map the event to the corresponding abstract action. Fig. 1(a) illustrates this operation.

(b) Generalization. KarDo then performs static analysis on the abstract actions in each recorded trace to eliminate irrelevant actions that do not affect the final system state. Once it has the relevant actions for each task, it proceeds to generalize them to deal with diverse configurations. Since navigation actions do not update state, KarDo can learn the many diverse ways to navigate the GUI from *totally unrelated tasks*, and therefore builds a global navigation graph across all tasks. In contrast, for state-modifying actions (i.e., UPDATES and COMMITS), KarDo uses differences across recordings of the *same task* to learn the different sequences of state-modifying actions that perform the task on various configurations, and represents this knowledge as a per task directed graph parameterized by configuration. Fig. 1(b) illustrates the generalization stage.

(c) Replay. In order to perform the task in a specific environment, KarDo walks down the graph of state-modifying actions trying to find a branch where all the actions involve applications (i.e. Thunderbird, Firefox, etc.) that exist on the machine. Once it finds such a branch, it proceeds to execute the actions along it. It moves from one state modifying action to the next by leveraging the global navigation graph to find a path from one of the active desktop widgets to the widget corresponding to the next state-modifying action. Fig. 1(c) illustrates the replay stage.

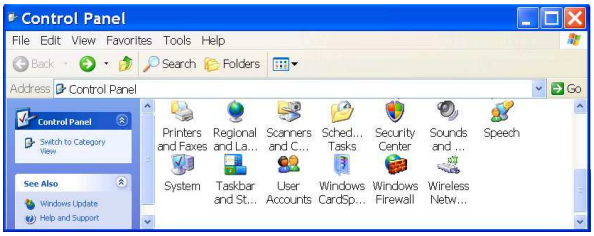
We built a prototype of KarDo as a thin client connected to a cloud-based service. We evaluate KarDo on 57 computer tasks drawn from the Microsoft Help website [9] and the eHow [4] websites which together include more than 1000 actions and include tasks like configuring a firewall, web proxy, and email. We generate a pool of 20 diversely configured virtual machines which we separate into 10 training VMs and 10 test VMs. For each task, two users performed the task on two randomly chosen VMs from the training set. We then attempt to perform the task on the 10 test VMs. Our results show that a baseline that tries both user traces on each test VM, and picks whichever works better, succeeds in only 18% of the cases. In contrast, KarDo succeeds on 84% of the 500+ VM-task pairs. Thus, KarDo can automate computer tasks across a wide variety of configurations without modifying the OS or applications.

We also performed a user study on 5 different computer tasks, to evaluate how well KarDo performs compared to humans for the same set of tasks. Even with detailed instructions from our lab website the students failed to correctly complete the task in 20% of the cases. In contrast, when given traces from all 12 users, KarDo produced a correct canonical solution which played back successfully on a variety of different machines.

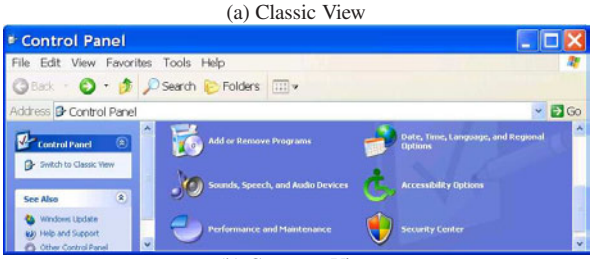
2 Challenges

A system that aims to automate computer tasks based on user executions and without instrumenting the OS or applications, needs to attend to multiple subtleties.

(a) Generalizing Navigation. Consider the task of configuring a machine for access through remote desktop. On Microsoft Windows, the first step is to enable remote desktop on the local machine through the "System" dialog box which is accessed through the Control Panel. Automatically navigating to this dialog box can be difficult however because the Control Panel can be configured in three different ways. Novice users typically retain the de-



(a) Classic View



(b) Category View

Figure 2: **Diverse Configurations.** To enable remote desktop one must go to the "System" dialog box. Depending on the configuration of the Control Panel, one can either directly click the "System" icon (a) or must first navigate to "Performance and Maintenance" (b) then click the "System" icon.

fault view which uses a category based naming scheme, as in Fig. 2(a). Most advanced users however switch to the "Classic View" which always shows all available controls, as in Fig. 2(b). And, efficiency oriented users often go as far as configuring the control panel so it appears as an additional menu off of the start menu. All three paths however lead to the same "System" dialog box where one can turn on remote desktop. The challenge is to produce a canonical GUI solution that performs the task on machines with any of these configurations even when the recorded traces for this task show only one of the possible configurations.

(b) Filtering Mistakes and Irrelevant Actions. KarDo needs a mechanism to detect mistakes and eliminate irrelevant actions that are not necessary for the task. For example, while performing a task, the user may accidentally open some program that turns out to not be relevant for the task. If this mistake is included in the final solution, however, it will require the playback machine to have this irrelevant program installed in order for KarDo to automatically perform the task. It is important to remove mistakes like this to prevent the need for the user to rerecord a second "clean" trace, thus allowing users to generate usable recordings as part of their everyday work.

(c) Parameterizing Replay. After enabling remote desktop on his local machine, the user needs to configure the router to allow through the incoming remote desktop connections and direct them to the right machine. KarDo can easily automate a task like this, since it is done through a web-browser interface to the router, which provides the same accessibility information as all other GUI applications. The challenge arises, however, because one user may have a static IP address while another has a dynamic IP address, or worse, one user might have a DLink router, while another has a Netgear. Different steps are required to perform this task if the user has a static IP address vs. a dynamic IP address. Similarly, different routers present different web-based configuration interfaces, so users with different routers need to perform different GUI actions to perform this task. KarDo needs to retain each of these paths in the final canonical solution, and parametrize them such that the appropriate path can be chosen during playback. The challenge is to distinguish these configuration based differences from mistakes and irrelevant actions so that the former can be retained while the later are removed.

(d) User-Specific Entries. Some tasks require a user to enter his name, password, or other user-specific entries. KarDo can easily avoid recording passwords by recognizing that the GUI naturally obfuscates them, providing a simple heuristic to identify them. However, KarDo also needs to recognize all other entries that are user specific and distinguish them from entries that differ across traces because they are mistakes or configuration-based differences. It is critical to distinguish user specific entries from mistakes and configuration differences because KarDo should ask the user to input something like his username, while it should automatically discover which path to follow for different router manufacturers.

3 KarDo Overview

KarDo is a system that enables end users to automate computer tasks without programming, and does not require modifications to the OS or applications. It has two components, a client that runs on the user machine to do recording and playback, and a server that contains a database of solutions.

When a user performs a task that he thinks might be useful to others, he asks the KarDo client to record his windowing events while he performs the task. If the user cannot, or does not want to perform the task on his machine, he can perform the task remotely on a virtual machine running on the KarDo server, while KarDo records his windowing events. In either case, when the user is done, the client uploads the resulting windowing event trace to the KarDo server. The server asks the user for a task name and description. It uses this information to search its database for similar tasks and asks the user if his task matches any of those. This ensures that all traces for the same task are matched together.

When a user encounters a task he does not know how to perform, he searches the KarDo database for a solution. KarDo's search algorithm has access to not only the information that a normal text search would have, such as the task's name and description, the steps of the task,

and the text of each relevant widget, but also system level information like which programs are installed, and which GUI actions he has taken recently. As a result, we believe that task search with KarDo can be much more effective than standard text searching is today. However, effective search represents a research paper on its own, and so we leave the search algorithm details to future work.

The user can either use the solution as a tutorial that will walk him through how to perform the task step by step, or allow the solution to automatically perform the task for him. It is important to recognize however that KarDo's solutions are intended to be best-effort. Even a highly evolved system will not be able to automate correctly all of the time. Thus, KarDo takes a Microsoft Virtual Shadow Service snapshot before automatically performing any task, and immediately rolls back if the user does not confirm that the task was successfully performed (as discussed in §8, however, we leave the security aspects of this problem to future work).

The next three sections detail the three steps for transforming a set of traces recorded on one set of machines into a solution which allows automated replay on any other machine. §4 covers how to record the windowing events and map them to abstract actions that highlight how each action affects the system state. §5 then describes how to merge together multiple such sequences of abstract actions to create a generalized solution for any configuration. Finally, §6 discusses how replay utilizes the generalized solution and the state of the playback machine to determine the exact set of playback steps appropriate for that machine.

4 Windowing Events to Abstract Actions

The first phase of generating a canonical solution from a set of traces is to transform a windowing event trace into a sequence of abstract actions, since the generalization phase, discussed in §5 works over abstract actions. Performing this abstraction requires first converting the trace to a sequence of raw GUI actions by associating GUI context information with each windowing event, and then mapping raw GUI actions to abstract actions using a machine learning classifier.

4.1 Capturing GUI Context

A low-level windowing event contains only the specific key pressed, or the mouse button click along with the screen location. Effectively mapping these low-level events to abstract actions requires additional information about the GUI context in which that event took place such as which GUI widget is at the screen location where the mouse was clicked. KarDo gathers this information using the Microsoft Active Accessibility (MSAA) interface [8]. Developed to enable accessibility aids for users with impaired vision, the accessibility interface has been built into all versions of the Windows platform since Windows 98 and is now widely supported [8]. Apple's OS X already provides a similar accessibility framework [7], and the Linux community is working to standardize a single accessibility interface as well [5]. The accessibility interface provides information about all of the visible GUI widgets, including their type (button, list box, etc.), their text name, and their current value, among other characteristics. It also provides a naming hierarchy of each widget which we use to uniquely name the widget. KarDo uses this context information to transform each windowing event to a raw GUI action performed on a particular widget. An example of such a raw GUI action is a left click on the OK button in the Advanced tab in the "Internet E-mail Setting" window.

4.2 Abstract Model

KarDo uses an abstract model for GUI actions. This model captures the impact that each action has on the underlying system state. We do not claim that our model captures all possible applications and tasks, however, it does capture common tasks (e.g., installation, configuration changes, network configurations, e-mail, web tasks) performed on typical Windows applications (e.g., MS Office, IE, Thunderbird, FireFox) as shown from the 57 evaluation tasks in Table 3. As discussed in §12, it also can be extended if important non-compliant tasks or applications arise.

In the abstract model all actions are performed on *widgets*. A widget could be a text box, a button, etc. There are three types of abstract actions in KarDo's model:

UPDATE **Actions:** These actions create a pending change to the system state. Examples of UPDATE actions include editing the state of an existing widget, such as typing into a text box or checking a check-box, and adding or removing entries in the system state, e.g., an operation which adds or removes an item from a list-box.

COMMIT/ABORT **Actions:** These actions cause pending changes made by UPDATE actions to be written back into the system state. An example of a COMMIT action is pressing the OK button, which commits all changes to all widgets in the corresponding window. An ABORT action is the opposite: it aborts any pending state changes in the corresponding window, e.g., pressing a Cancel button.

NAVIGATE **Actions:** These change the set of currently visible widgets. NAVIGATE actions include opening a dialog box, moving from one tab to another, or going to the next step of a wizard by pressing the Next button.

Note that a single raw GUI action may be converted into multiple abstract actions. For example, pressing the

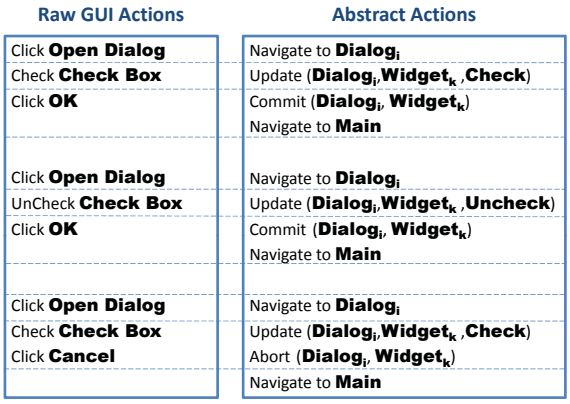


Figure 3: **A simplified illustration mapping raw GUI action to the corresponding abstract actions.**

OK button both commits the pending states in the corresponding window and navigates to a new view.

Fig. 3 illustrates a simple sequence of raw GUI actions and the corresponding abstract actions. Here a user clicks to open a dialog box, clicks to check a check box, and then clicks OK. He then realizes that he made a mistake and opens the dialog again to uncheck the check box. Finally, he opens the dialog one last time, rechecks the box, but reconsiders his change and hits the Cancel button. The corresponding sequence of abstract actions shows that the user navigated thrice to the dialog box, updated the check box, committed or aborted the UPDATE, and navigated again to the main window. However, the abstract model allows us to reason that the first UPDATE and the corresponding NAVIGATE and COMMIT actions are overwritten by the later UPDATE and hence are redundant and can be eliminated. Similarly, since the last UPDATE and associated ABORT do not update the state, they too can be eliminated. In §5.1, we describe KarDo's static analysis algorithm for filtering out such mistakes.

4.3 Mapping to Abstract Actions

KarDo has to label the raw GUI actions returned by the accessibility interface as UPDATE, COMMIT, and/or NAVIGATE. It does not attempt to explicitly classify ABORT actions because KarDo's algorithms implicitly treat the lack of a COMMIT action as an ABORT action as explained in §5.1. Further, a given action can have multiple different abstract action labels, or not have any label at all. KarDo performs the labeling as follows.

To label an action as a NAVIGATE action, KarDo uses the simple metric of observing whether new widgets become available before the next raw action. Specifically, KarDo's recordings contain information about not only the current window, but all other windows on the screen. Thus, if an action either changes the available set of widgets in the current window, or opens another window, then KarDo labels that action as a NAVIGATE action.[1]

Widget/ Window Features	Widget name (typically the text on the widget) Widget role (i.e., button, edit box, etc.) Does the widget contain a password? Is the widget updatable (i.e., check box, etc.)? Is the widget in a menu with checked menu items? Does the window contain an updatable widget?
Response To Action Features	Did the action cause a window to close? Did the action cause a window to open? Did the action generate an HTTP POST? Did the action cause the view to change? Did the action cause the view state to change?
Action Features	Action type (right mouse click, keypress, etc.) Keys pressed (the resulting string) Does the keypress contain an "enter"? Does the keypress contain alpha numeric keys? Is this the last use of the widget?

Table 1: **SVM Classifier Features.** This table shows the list of features used by the SVM classifier to determine which actions are UPDATE and COMMIT actions. All features are encoded as binary features with multi-element features (such as widget name) encoded as a set of binary features with one feature for each possible value.

Labeling an action as a COMMIT or UPDATE action is not as straightforward. There are cases where this labeling is fairly simple; for example, typing in a text box or checking a check box is clearly an UPDATE action. But to handle the more complex cases, KarDo approaches this problem the same way a user would, by taking advantage of the visual features on the screen. For example, a typical feature of a COMMIT action, is that it is associated with a user clicking a button whose text comes from a small vocabulary of words like {OK, Finish, Yes}.

KarDo does this labeling using a machine learning (ML) classifier. Specifically, an ML classifier for a given class takes as input a set of data points, each of which is associated with a vector of features and produces as output a label for each data point indicating whether or not it belongs to that class. It does this labeling by learning a set of weights which indicate which features, and which combinations of features, are likely to produce a positive data point, and which are likely to produce a negative data point. KarDo uses a supervised classifier, which does this learning based on a small set of training data.

KarDo uses two separate classifiers, one for COMMITS and one for UPDATES. These classifiers take as input a data point for each user action (i.e., each mouse click or keypress), and label them as UPDATES and COMMITS respectively. [2] Table 1 shows the features used by KarDo's classifiers to determine the labels. Features such as widget name, and widget role cannot be used directly by the classifiers however, because classifiers only work with numerical features. Thus, KarDo handles features

[1] KarDo will also label a window close as a NAVIGATE action in cases like a modal dialog box, where the user cannot interact with the underlying window again until the dialog box is closed.

[2] Note that since a given action is fed to both classifiers it can be classified as both an UPDATE and a COMMIT to account for actions like clicking the "Clear Internet Cache" button which both update the state and immediately commit that update.

like these, which are character strings, using the same technique as the Natural Language Processing community. Specifically, it adds a new binary feature for each observed string, i.e., is the the widget name "OK", is the widget name "Close", etc. This creates a relatively large number of features for each action which can cause a problem called overfitting, where the classifier works well only on the training data set, and it does not generalize to new data. To handle this large number of features, KarDo uses a type of classifier called a Support Vector Machine (SVM) which is robust to large numbers of features because it uses a technique called margin maximization. KarDo trains the SVM classifier using a set of training data from one set of traces, while all testing is done using a distinct set of traces.

5 Generalization

Generalization starts with multiple abstract action traces which perform the same task on different configurations and transforms them into a single canonical solution that performs the task on all configurations. KarDo performs this step by separating how it handles NAVIGATE actions from how it handles state modifying actions, i.e. UPDATES and COMMITS. Specifically, it first prunes out all NAVIGATE actions from each trace (and all unlabeled actions), leaving only the state modifying actions. It then follows a three step process to generate a canonical solution: (1) it runs a static analysis algorithm on each pruned trace that removes all the mistakes and irrelevant UPDATES; (2) these simplified traces are merged together to create a single canonical trace which is parameterized by user-specific environment; and (3) the NAVIGATE actions from all traces for all tasks are utilized to create a global navigation graph which is used to do navigation during playback. The rest of this section describes these three steps in detail.

5.1 Filtering Mistakes

The first step of generalization is to filter out mistakes from each trace. To understand the goal of filtering out mistakes, consider the example in Fig. 3, where the user opens the dialog box multiple times, changing the value of a given widget each time. In this example, the first check box UPDATE is overwritten by the second, while the third is never committed. Thus both of these UPDATES are unnecessary, and they should be removed along with the opening and closing of the dialog box associated with them. Their removal is important for two reasons. First, if a user chooses to read the text version of a solution, or to have KarDo walk him through the task, then such mistakes will be confusing to the user. Second, if not removed, mistakes like this can be confused as

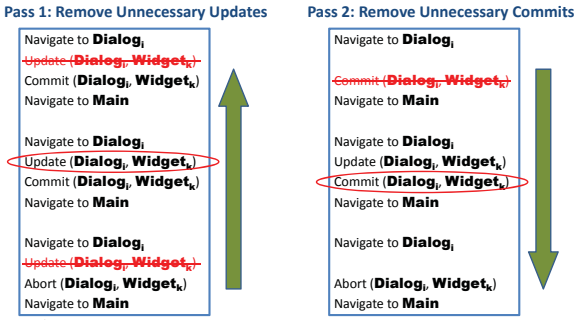


Figure 4: **A Two-Pass Algorithm to Remove Mistakes.**

user-specific or environment-specific actions and hence limit our ability to generalize.

The naive approach to identifying mistakes would compare multiple GUI traces from users who performed the same task, and consider differing actions as mistakes. Unfortunately, such an approach will also eliminate necessary actions which differ due to differences in users' personal information (e.g., printer name) or their working environment (e.g., different wireless routers).

In contrast, the key idea in KarDo is to recognize that the difference between unnecessary actions and environment specific actions is that unnecessary actions do not affect the final system state, and GUIs are merely a way of accessing this system state. So KarDo tracks the state represented by each widget and keeps only actions that affect the final state of the system. It does this using the following two-pass static analysis algorithm that resembles the algorithms used in various log recovery systems to determine the final set of committed UPDATES.

Pass 1 - Filtering Out Unnecessary UPDATES: The first pass removes all UPDATES on a particular widget except the last UPDATE which actually gets committed. Specifically, consider again our example from Fig. 3 where a user opens a given dialog box, and modifies a widget three times. We can see that KarDo needs to recognize that the second UPDATE overwrote the first UPDATE, rendering the first unnecessary. However, it cannot blindly take the last UPDATE, because the final UPDATE was aborted. Thus KarDo needs to keep the final *committed* UPDATE for each widget. It does this by walking backwards through the trace maintaining both a list of outstanding COMMITS, and a list of widgets for which it's already seen a committed UPDATE. As it walks backwards, it removes both UPDATES without outstanding COMMITS and UPDATES for which it's already seen a committed UPDATE on that same widget.

Pass 2: Filtering Out Unnecessary COMMITS: The second pass removes COMMITS with no associated UPDATES. It does this by walking *forwards* through the trace maintaining a set of pending UPDATES. When it reaches an UPDATE, it adds the affected widget to the

pending set. When it reaches a COMMIT, if there are any widget(s) associated with this COMMIT in the pending set, it removes them from the pending set, otherwise it removes the COMMIT from the trace.

One may fear that there are cases in which having the system go through an intermediate state is necessary even if that state is eventually overwritten. For example, if the task involves disabling a webserver, updating some configuration that can only be modified when the webserver is disabled and then re-enabling the webserver, it would be incorrect to remove the disabling and re-enabling of the webserver. While in theory such problems could arise, we find that in practice they do not arise. This is because actions like enabling and disabling a webserver typically look to KarDo like independent UPDATES which do not reverse each other, since one may require clicking the "disable" button while the other requires clicking the "enable" button. This causes the mistake removal algorithm to be somewhat conservative, which is the appropriate bias since it's worse to remove a required action than to leave a couple of unnecessary actions.

5.2 Parametrization

The second step of generalization is to parameterize the traces. Specifically, now that we have removed mistakes and navigation actions, the remaining differences between traces of the same task are either user specific actions (e.g. user name), or machine configuration differences (static IP vs. dynamic IP) which change the set of necessary UPDATE or COMMIT actions. To integrate these differences into a canonical trace that works on all configurations KarDo parametrizes the traces as follows:

(a) Parametrize UPDATES. The values associated with some UPDATE actions, such as usernames and passwords, are inherently user specific and cannot be automated. KarDo identifies these cases by recognizing when two different traces of the same task update the same widget with different values. To handle these kinds of UPDATES, KarDo parses all traces of a task to find all unique values that were given to each widget via UPDATE actions that were subsequently committed. Based on these values the associated UPDATE actions are marked as either *AutoEnter* if the associated widget is assigned the same value in all traces of that task, or *UserEnter* if the associated widget is assigned a different value in each trace. On play back, AutoEnter UPDATES are performed automatically, while KarDo will stop play back and ask the user for UserEnter actions. Note that if the widget is assigned to a few different values, many of which occur in multiple traces (e.g., a printer name), KarDo will assign it *PossibleAutoEnter*, and on play back let the user select among values previously entered by multiple different users or enter a new value.

(b) Parameterized Paths. All of the remaining differences between traces now stem from configuration differences in the underlying machine, which necessitate a different set of UPDATES or COMMITS in order to perform the same task. To handle this type of difference, KarDo recognizes that when a user's actions in two different traces differ because of the underlying machine configuration, the same action will generate two different resulting views. For example, consider the task of setting up remote desktop. Different traces may have used different routers, which require different sets of actions to configure the router. Since the routers are configured via a web browser, opening a web browser and navigating to the default IP address for router setup, http://192.168.1.1, will take the user to a different view depending on which router the user has. KarDo takes advantage of this to recognize that if the DLink screen appears, then it must follow the actions from the trace for the DLink router, and similarly for the other router brands.

Thus, KarDo builds a per-task state-modifying graph and automatically generates a separate execution branch with the branch point parameterized by how the GUI reacts, e.g., which router configuration screen appears. This ensures that even when differences in the underlying system create the need for different sets of UPDATES and COMMITS, KarDo can still automatically execute the solution without needing help from the user. If the traces actually perform different actions even though the underlying system reacts exactly the same way, then these are typically mistakes, which would be removed by our filtering algorithm above. If differences still exist after filtering, this typically represents two ways of performing the same step in the task, i.e. downloading a file using IE vs. Firefox. Thus KarDo retains both possible paths in the canonical solution and if both are available on a given playback machine, then KarDo will choose the path that is the most common among the different traces.

5.3 Building a Global Navigation Graph

Real world machines expose high configuration diversity. This diversity stems from basic system level configuration like which programs a user puts on their desktop and which they put in their Start Menu, to per application configuration like whether a user enables a particular tool bar in Microsoft Word, or whether they configure their default view in Outlook to be e-mail view or calendar view. All of these configuration differences affect how one can reach a particular widget to perform a necessary UPDATE or COMMIT. KarDo handles this diversity with only a few traces for each task by leveraging that multiple tasks may touch the same widget, and building a single general navigation graph using traces for *all tasks*.

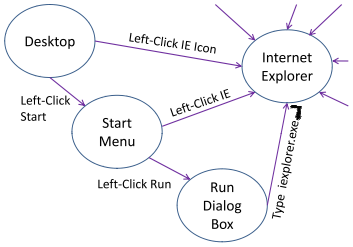


Figure 5: **Illustration of the Navigation Graph.** A simplified illustration showing a few ways to reach the IE window. The actual graph is per widget and includes many more edges.

Building such a general navigation graph is relatively straightforward. KarDo marks each NAVIGATE action as an enabler to all of the widgets that it makes available. KarDo then adds a link to the navigation graph from the widget this NAVIGATE action interacted with (e.g., the icon or button that is clicked), to the widgets it made available, and associates this NAVIGATE action with that edge. Fig. 5 presents a simplified illustration of a portion of the navigation graph. It shows that one can run IE from the desktop, the Start menu, or the Run dialog box.

6 Replay

The replay process takes a solution constructed using the process described in the preceding sections, and produces the low-level window events to perform a task on a particular machine. At each step, this process utilizes the full navigation graph, the per-task state-modifying dependency graph, and the current GUI context.

During replay, KarDo walks down the task's state-modifying dependency graph. As described in §5.2, this graph is parameterized by GUI context. Thus, KarDo utilizes the current GUI context and the installed applications to determine the path to follow at any branch point.

At each step, KarDo needs to ensure that the next state-modifying action is enabled. To enable a given UPDATE/COMMIT action, KarDo finds the shortest directed path in the navigation graph between the widget required for the UPDATE/COMMIT action, and any widget that is currently available on the screen. KarDo finds this path by working backwards in the navigation graph. Specifically, it first checks to see if the necessary widget is already available. If not, it looks in the navigation graph for all incoming edges to the necessary widget, and checks to see if any of the widgets associated with those edges are available. If not, it checks the incoming edges to those widgets, etc. It continues this process until either it finds a widget which is already available on the screen, or there are no more incoming edges to parse.

Once KarDo's navigation algorithm finds a relevant widget in the navigation graph which is currently available on the screen, it performs the associated action. If the expected next widget in the graph appears, KarDo follows the path through the navigation graph until the widget associated with the necessary UPDATE/COMMIT action becomes available. If at any point, the expected widget that the edge leads to does not appear, however, KarDo marks that navigation edge as unusable, and again performs the above search process. [3] It continues this process until either it succeeds in making the necessary UPDATE/COMMIT widget appear on the screen, or it has exhausted all possibilities and has no paths left in the navigation graph between widgets currently on the screen and the next necessary UPDATE/COMMIT widget.

Finally, each abstract action, whether NAVIGATE or state-modifying, is mapped to a low-level windowing event by utilizing the accessibility interface similar to the way it is used during recording.

7 Solution Validation

When a user uploads a solution for a task, KarDo allows the user to provide a solution-check. To do so, the user performs the steps necessary to confirm the task has been completed correctly and highlights the GUI widget that indicates success. For example, to check an IPv6 configuration, the user can go to ipv6.google.com and highlight the Google search button. As with standard tasks, KarDo will map the trace to abstract actions, clean it from irrelevant actions, etc. Such solution-checks allow KarDo to confirm that its canonical solution for a task works on all configurations by playing the solution followed by its solution-check on a set of VMs with diverse configurations, and checking that in each VM the highlighted GUI widget has the same state as in the solution-check.

8 Security

Ensuring that users cannot insert malicious actions into KarDo's solutions is an important topic that represents a research paper on its own. We do not attempt to tackle that problem in this paper. To handle non-malicious mistakes, however, KarDo takes a Microsoft Virtual Shadow Service snapshot before automatically performing a task and rolls back if the user is unhappy with the results.

9 Implementation

The KarDo implementation has three components: a client for doing the recording and the playback, a server to act as the solution repository, and a virtual machine infrastructure for remote recording and solution testing.

9.1 Client

Our current KarDo client is built on Microsoft Windows as a browser plugin. The user interface runs in the

[3] It caches the searched subgraphs to speed up any later searches.

browser and is built using standard HTML and Javascript which communicate with the plugin to provide all KarDo functionality. The plugin is written natively in C++. As discussed in §4.1, the plugin uses the OS Accessibility API to do the recording and playback.

The main implementation challenge in the client is to ensure that the GUI context of each mouse click and keypress can be recorded before the GUI changes as a result of the user action, i.e. before the window closes as a result of clicking the "OK" button. KarDo achieves the timely recording of the GUI context by utilizing the Windows Hooks API, which allows registration of a callback function to be called immediately before keypress and/or mouse click messages are passed to the application. The challenge is that such a callback function needs to be extremely fast, otherwise the UI feels sluggish to the user [17]. Calls to MSAA to get the GUI context are very slow, however, for two reasons: (1) they use the Microsoft Component Object Model (COM)[4] interface to marshal and unmarshal arguments for each function call, and (2) MSAA requires a separate call for each attribute of each widget on the screen (e.g., a widget name or role) often resulting in thousands of COM function calls per window.

We use two main techniques to maintain acceptable recording performance. First, we implement the callback function in a shared library so that it can run in-process with the application receiving the click/keypress. This significantly improves performance since it avoids the overhead of COM IPC for each function call. Second, instead of recording the GUI context of every window on the screen with every user input, we record only the full context of the window receiving the user input, and for all other windows we record only high level information such as the window handle, and window title. As we show in §10.4, this significantly improves performance when the user has many other windows open.

9.2 Solution Repository Server

The solution server provides a central location for upload, download and storage of all solutions. In our current implementation, all solution merging also happens on the server. We implement the solution server on Linux using a standard Apache/Tomcat server backed by a Postgres database. All solutions are stored on disk, with all meta-data stored in the database. When the client finishes recording a trace, KarDo immediately asks the user if he would like to upload the trace. Upon confirmation, the client uploads the trace to the server, and the server searches its existing database for solutions with similar sets of steps, and asks the user to confirm if his trace matches any of these. The server also provides a web interface listing all solutions. When a user finds a task they would like automatically performed, they click the *Play* button which calls into the client browser plugin to download that solution from the server and start playback.

9.3 Virtual Machine Infrastructure

The VM infrastructure is used for two purposes: 1) to enable users to record a solution for a task which they either cannot or do not want to perform on their own machine; and 2) to perform solution validation as discussed in §7.[5] KarDo's VM infrastructure is build on top of Kernel-based Virtual Machine (KVM)[6]. Its design is based on Golden Master (GM) VM images, which are generic machine images that have been configured to expose a certain dimension of configuration diversity, or make available a certain set of tasks. For example, some GMs are configured with static IP addresses, while others have dynamic IP addresses, and some have Outlook as the default mail client, while others have Thunderbird. The infrastructure can then quickly bring up a running snapshot of any GM by taking advantage of KVM's copy-on-write disks and its memory snapshotting support.

10 Evaluation

We evaluate KarDo on 57 computer tasks which together include more than 1000 actions and are drawn from the Microsoft Help website [9] and the eHow [4] website. We chose these tasks by randomly pulling articles from the websites and then eliminating those which did not describe an actual task (i.e. "What does Microsoft Exchange do?"), those which described hardware changes (i.e. "How to add more RAM"), and those which required software to which we did not already have a license. We focused on common programs, e.g., Outlook, IE, and diversified the tasks to address Web, Email, Networking, etc. The full list of tasks is shown in Table 3 and includes tasks like configuring IPv6, defragmenting a hard drive, and setting up remote desktop.

10.1 Handling Configuration Diversity

Our goal with KarDo is to handle the wide diversity of ways in which users configure their machines. Measuring KarDo's performance on a small number of actual user machines is not representative of the wide diversity of configurations, however, since many users leave the default option for most configuration settings. To capture this wide diversity, we generate a pool of 20 virtual machines whose configurations differ along the following axes: differences of installed applications (e.g., Firefox

[4] a binary interface used for inter-process communication

[5] We also used it to produce the evaluation results in §10.1.

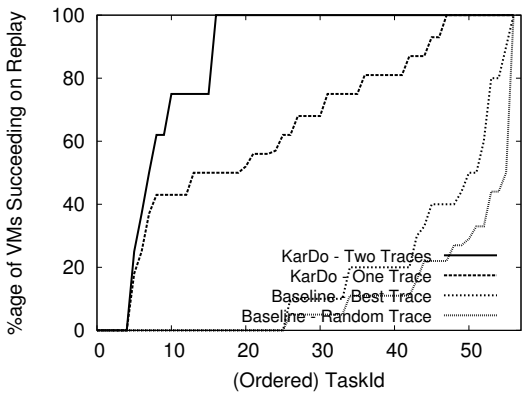


Figure 6: **Success Rate on Diverse Configurations:** For each task on the x-axis the figure plots on the y-axis the percentage of test VMs that succeeded in performing the task using a specific automation scheme. For each scheme, the area under the curve refers to the success rate taken over all task-VM pairs. KarDo-Two-Traces has a success rate of 84%, whereas KarDo-One-Trace has a success rate of 64%. In contrast, Best-Trace, which tries both of the two traces and picks whichever works better, has a success rate of only 18%, and Random-Trace, which randomly chooses between the two traces, has a success rate of only 11%.

vs. IE, Thunderbird vs. Outlook), differences of per-application configuration (e.g., different enabled tool and menu bars), user-specific OS configuration (e.g., different views of the control panel, different icons on the desktop), and different desktop states (e.g., different windows or applications already opened). We apply each configuration option to a random subset of the VMs. This results in a set of machines with more configuration diversity than normal, but which represent the kind of diversity of configurations we would like to handle.

We separate this pool of VMs into 10 training and 10 test. We recruited a set of 6 different users to help us record traces, including 2 non-expert users and 4 computer science experts. For each of the 57 evaluation tasks, two of the six users perform the task on two randomly chosen VMs from the training set. We then try to replay each task on the 10 test VMs. We compare four schemes:

- **KarDo - Two Traces:** We generate a canonical solution by merging together the two traces for each task, and we generate a navigation graph using all of the traces from all tasks. We then use the KarDo replay algorithm to playback the resulting solutions on the test VMs.
- **KarDo - One Trace:** We randomly pick one of the two traces and use it to generate a canonical solution for that task. The navigation graph is generated from that trace plus all traces for all other tasks (but not the other trace for that same task).
- **BaseLine - Best Trace:** For each VM, we try directly playing both of the two recorded traces for each task. If either trace succeeds then we report success for that VM-task combination. This shows how well a baseline system would perform with two traces per task.
- **BaseLine - Random Trace:** We randomly pick one of the two traces and directly playback all of the GUI actions in the original trace on the test VMs. This represents how well a baseline system would perform with only one trace per task.

Fig. 6 plots the success rate of these four schemes. It shows that the Best-Trace approach succeeds on average on only 18% of the VMs while the Random-Trace succeeds on just 11% of the test VMs. In contrast, KarDo succeeds on 84% of the 500+ VM-task pairs when given two traces, and on 64% when given only one trace. Thus, KarDo enables non-programmers to automate computer tasks across diverse configurations.

10.2 Understanding Baseline Errors

The Best-Trace and Random-Trace schemes are very susceptible to configuration differences. Even a single configuration difference can cause the Random-Trace scheme to fail. The two traces considered by the Best-Trace approach make it more robust to configuration differences, but it still only works if the test VM looks very similar to one of the VMs on which the recordings were performed. Consider a case where one recording opened Outlook from the desktop, and then accessed a menu item to change some configuration, and the other recording opened it from the Start Menu, and then used the tool bar to change that configuration. Even in this simple case where the two recordings see a large amount of diversity between them, the Best-Trace algorithm cannot handle a case where the tool bars are turned off, but Outlook is not on the desktop, or a case where menus are turned off, but Outlook is not in the Start Menu. More generally, even if the test VM is a hybrid of the two VMs on which the traces were recorded, the Best-Trace approach will fail. This is because a hybrid configuration requires pulling different parts from each of the traces which cannot be done without KarDo's technique of merging the traces together. Thus, the Best-Trace approach requires an excessive number of examples to successfully play back on diverse machines. Finally, we note that there are a number of tasks where the Best-Trace fails on all VMs. This occurs when all test VMs are widely different from the two VMs where the recordings were performed.

10.3 Understanding KarDo Errors

While KarDo successfully plays back in the vast majority of the cases, it still fails to playback successfully on 16% of the VM-task pairs. There are three main causes of these errors: classifier mistakes, incorrect navigation steps, and missing navigation steps. Fig. 7 shows the breakdown of these errors. Specifically it shows that eliminating classification errors results in a 91% success

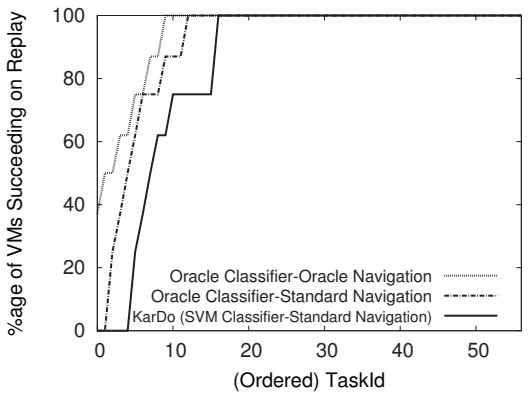


Figure 7: **Cause of KarDo Errors:** This figure shows the breakdown of the KarDo playback errors by showing the playback success when various parts of the KarDo algorithms are replaced by oracle versions. Recall that the success rate is the area under the curve. Based on the figure, replacing KarDo's classifiers with oracle classifiers increases the playback success rate from 84% to 91%. Additionally, eliminating all mistakes in the navigation database by using an oracle for navigation increases the playback success rate from 91% to 95%. The remaining 5% failure cases result from missing navigation steps that did not appear in any of the input traces.

rate while eliminating incorrect navigation steps results in a 95% success rate. We observe that the remaining 5% of the errors result mostly from missing navigation steps. The following discusses each of these in detail.

(a) ML Classification Errors: To evaluate our ML classifier, we manually labeled each of the actions performed by the users for the 57 tasks as a COMMIT action, an UPDATE action, both or neither. We then split this labeled data into half training and half test data. As described in §4.3 we run two separate classifiers on the data, one for UPDATE actions, and one for COMMIT actions. Since KarDo's generalization algorithm (from §5) retains only COMMITS and UPDATES as necessary actions, false negative misclassifications will cause KarDo to skip one of these necessary UPDATES or COMMITS during playback. False positives on the other hand will cause unnecessary actions to be retained, requiring KarDo to attempt to playback irrelevant actions which may be unavailable on a test VM. We calculate the false positive rate for each of the two classifiers as the percentage of actions in the COMMIT/UPDATE class that should not be in it, and the false negative rate as the percentage of actions not in the COMMIT/UPDATE class but should be in it.

The resulting performance of the KarDo classifiers is shown in Table 2. As we can see, the ML classifiers perform quite well even though classification mistakes account for almost half of the playback failures. Specifically, the COMMIT classifier has a false positive rate of only 2% and a false negative rate of only 3%. The COMMIT classifier performs so well because COMMITS follow very predictable patterns, i.e., they almost always occur when a button is pressed, and very frequently cause the associated window to close. The UPDATE classifier performs slightly worse with a 6% false positive rate and a 5% false negative rate. The higher false positive rate for UPDATES is caused by actions using widgets like combo boxes and edit boxes which are typically used for UPDATES, but are sometimes used just for navigational purposes. Occasionally when an action uses one of these widgets only for navigation (i.e., it's not an UPDATE), KarDo will misclassify the action as an UPDATE action. The higher false negative rate stems from actions which are both UPDATES and COMMITS. These actions tend to look much more like COMMITS than UPDATES and as a result the COMMIT classifier typically correctly classifies them, but the UPDATE classifier occasionally misclassifies them, not realizing they are also UPDATES. One such example is clicking the button to defragment your hard drive, which looks very much like a COMMIT action as it is a button click, and closes the associated window, but does not look very much like a typical UPDATE action since button clicks usually do not update any system state. In fact, if we test the UPDATE classifier after removing actions that are both COMMITS and UPDATES from the training and test sets the false negative rate drops to 2% without increasing the false positive rate at all.

	False Positive Rate	False Negative Rate
COMMITS	2%	3%
UPDATES	6%	5%

Table 2: **Performance of the COMMIT and UPDATE Classifiers.**

Note that a misclassification does not necessarily cause an error in the resulting canonical trace. In particular, only misclassifications that result in the eventual discard of a necessary action produce erroneous task solutions. For example, one may misclassify an action that is both COMMIT and UPDATE as only COMMIT. Still, as long as the mistake removal algorithm keeps this action as necessary, the resulting solution will still perform the UPDATE.

To evaluate the effect of classification mistakes on the final playback performance, we ran an "Oracle Classifier" version of KarDo where instead of using the output from the ML classifier to determine whether an action is an UPDATE or a COMMIT, we directly use the hand generated labels so that all classifications are correct. As shown in Fig. 7 this increases the playback success rate by an additional 7%. More training data would help eliminate these mistakes.

(b) Incorrect Navigation Steps: The next cause of playback problems comes from limitations in the way we currently generate the navigation graph. As discussed in §4.3, KarDo assumes that navigation depends only on the final action that made a widget visible. In a few cases, however, navigation depends on other earlier actions in the trace. A simple example of this is the "Run" dialog box which allows a user to type in the name of a program and then click "OK" to run it. In this case, the naviga-

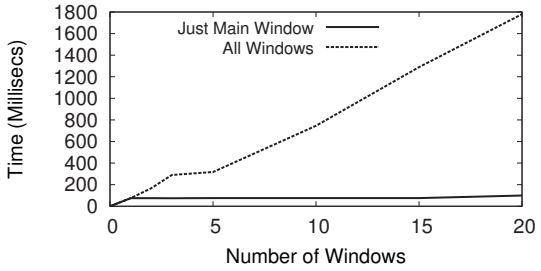


Figure 8: **Real Time Window System Context Recording:** The figure shows that KarDo's optimized recording, which limits recording full context information to only the main window, has a response time less than 100ms regardless of the number of windows. This is significantly below the 200ms threshold at which users perceive the UI to be sluggish. In contrast, recording the full context of all windows has a response time that scales with the number of windows, eventually becoming very slow.

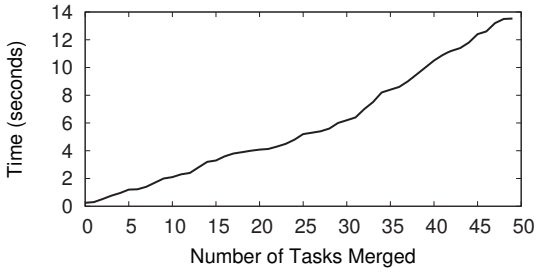


Figure 9: **Performance of Solution Merging:** The figure graphs the time that KarDo takes to merge a given number of traces, showing that KarDo can scale to quickly merge a large number of traces for a given task.

tion depends not only on clicking "OK", but also on the program name filled into the edit box.

To test the effect of incorrect navigation steps on the final playback success, we hand labeled all such dependent navigation actions. We then ran a "Oracle Navigation" version of KarDo where each navigation step had the full set of required actions associated with it. As shown in Fig. 7 this increases the playback success by an additional 4%. These mistakes can be eliminated by the additional classifier discussed in §12.

(c) Missing Navigation Steps: The final cause of playback problems stems from KarDo's fairly limited view of the GUI navigation landscape, due to the relatively small number of input traces in our experiments. Specifically, since many of the traces KarDo uses to generate its solutions are performed by users that already know how to perform a task, these traces rarely include navigation information related to incorrect navigations. This can cause playback to fail in the small fraction of cases where KarDo navigates in a way that is not appropriate for a given configuration and thus results in an error dialog box or some other GUI widget/window which was not seen in any trace. In this case, to ensure that it does not cause any problems, KarDo will immediately abort playback and roll back the user's machine to its original state. These type of errors account for most of the remaining 5% of playback errors shown in Fig. 7, and can be solved by more traces.

10.4 Feasibility Micro-Benchmarks

We want to ensure that KarDo's design performs well enough to be feasible in practice. To test this, we ran three performance tests on a standard 2.4 GHz Intel Core2 Duo desktop machine.

First, as discussed §9.1, context recording has to be fast so that it does not cause the user to perceive the UI as sluggish. Fig. 8 shows that even with many windows on the screen, KarDo can grab the relevant windowing system context in well less than 100ms, and the overhead is relatively constant regardless of the number of windows. Since users only start to notice delay when it is greater than 200ms [17], this additional delay should be acceptable to users. In contrast a scheme which records the context of all windows reaches an unacceptable delay of more than 1 second with even just 15 windows open.

Next, we check the performance of solution merging. Fig. 9 shows that merging up to 50 traces takes only 15 seconds, and it takes less than a second to merge 5 traces. This result shows that KarDo can easily scale to merging a large number of traces for each task.

Finally, KarDo's playback is relatively fast. For the 57 tasks in Table 3, playing a KarDo solution takes on average 52 seconds with a standard deviation of 9 seconds. The maximum replay time was 125 seconds, which was mostly spent waiting for the virus scanner to finish.

10.5 Working with Users

We evaluate KarDo's ability to improve on the status quo of using text instructions to perform computer tasks. We asked 12 CS students to perform 5 computer tasks within 1 hour, based on instructions from our lab website. We also used KarDo to automate each task by merging the students' traces into a single canonical solution.

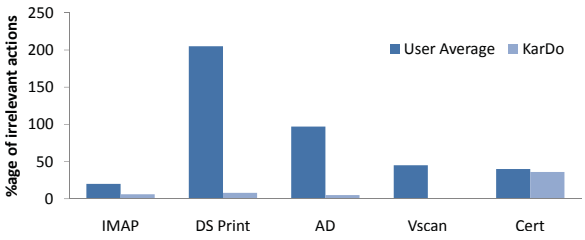
We find three important results. First, as shown in Fig. 10(a), even with detailed instructions, the students fail to correctly complete the tasks in 20% of the cases. In contrast, KarDo always succeeded in generating a solution that automated the task on all 12 user machines.

Second, as shown in Fig. 10(b), even when the students did complete the tasks they performed on average 84% more GUI actions than necessary, and sometimes more than three times the necessary number of actions. KarDo's automation removes most of these irrelevant actions, performing only 11% more actions than necessary.

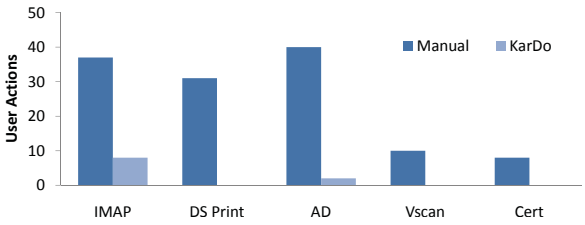
Third, as shown in Fig. 10(c), KarDo reduced the per-task required number of times the user had to interact with the machine from 25 to 2 times, on average. This reduction is because KarDo requires manual entry only for user-specific inputs, and automates everything else.

	IMAP Configuration	Double-Sided Printing	Active Directory	Virus Scan	Fix FireFox Certificate
KarDo	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
User 1	Yes	Yes	Yes	Yes	Yes
User 2	Yes	No	Yes	Yes	Yes
User 3	Yes	Yes	Yes	Yes	Yes
User 4	Yes	No	No	Yes	Yes
User 5	No	No	Yes	Yes	Yes
User 6	No	Yes	No	Yes	Yes
User 7	Yes	Yes	Yes	Yes	Yes
User 8	Yes	No	Yes	Yes	Yes
User 9	Yes	No	Yes	Yes	Yes
User 10	No	No	Yes	Yes	Yes
User 11	Yes	Yes	Yes	Yes	Yes
User 12	Yes	Yes	Yes	Yes	Yes

(a) Task successes and failures.



(b) Percentage irrelevant actions performed by users and KarDo



(c) User manual inputs with and without KarDo.

Figure 10: **Working with Users:** The figures shows that (a) KarDo performs the task correctly, even when many users fail, (b) KarDo filters most irrelevant actions, and (c) with KarDo users need to manually perform very few steps, typically only those which require user-specific information.

These results show that KarDo can help users reduce the time and effort spent on IT tasks.

11 Related Work

While there are many tools to help automate computer tasks, most either do not support recording and must be scripted by programmers (e.g., AutoIt [2] and AutoHotKey [1]), or allow recording only by relying on application specific APIs and thus cannot be used to automate generic computer tasks (e.g., macros, DocWizards [14]). Apple's Automator [3], Sikuli [13] and AutoBash [18] are the only exceptions as far as we know. However, neither Automator nor Sikuli can automatically produce a canonical GUI solution that works on different machine configurations. AutoBash covers only tasks which are entirely contained on the local machine, which is increasingly infrequent with today's networked computer systems. Additionally, it requires modifying the kernel to track dependencies across applications and then taking diffs of the affected files. Such kernel modifications are a deployment barrier, and file diffs are ineffective on binary file formats.

Some tools support recording and check pointing, such as DejaView [16], but they do not actually playback a task, instead only returning to a checkpointed state.

Lastly, there are tools that leverage shared information across a large user population [21, 20, 15, 19, 12, 11]. Strider [21] and PeerPressure [20] diagnose configuration problems by comparing entries in Windows registry on the affected machine against their values on a healthy machine or their default values in the population. FTN addresses the privacy problem in sharing configuration state by resorting to social networks [15]. [19] and [12] track kernel calls similar to AutoBash to determine problem signatures and their solutions. NetPrints [11] collects examples of good and bad network configurations, builds a decision tree, and determines the set of configuration changes needed to change a configuration from bad to good. All of these tools compare potentially problematic state information against a healthy state to address computer problems and failures. KarDo focuses on a complementary issue where the existing machine state maybe perfectly functional but the user wants to perform a new task. KarDo addresses such how-to tasks by working at the GUI level, which allows it to handle any general task the user can perform.

12 Addressing KarDo's Limitations

While our system represents a first step towards providing a system for automating a task by doing it, our current implementation has multiple limitations we expect to explore in future work. First, our model of labeling all actions as COMMITS, UPDATES and NAVIGATE actions is not exhaustive. Specifically, it does not cover tasks which simply show something on the screen. For example, a task like "Find my IP Address" will look to KarDo like it does nothing, and so all actions will be removed. This can be addressed by extending the model. Second, as discussed in §10.1, it does not handle tasks containing complex navigation actions. For example if navigation requires typing the name of a program in an edit box and then clicking "Run" then KarDo will only click the "Run" button. This can be solved using an additional classifier to detect these dependent navigation actions. Finally, KarDo requires unnecessary manual steps when entering the same user specific information across many tasks. For example, a user will have to manually enter his Google username every time he wants to run any task that accesses Google services.To handle this, we'd

like to build a profile for each user which will remember previous inputs by a user and reuse them across tasks.

13 Concluding Remarks

This paper presents a system for enabling automation of computer tasks, by recording traces of low-level user actions, and then generalizing these traces for playback on other machine configurations through the use of machine learning and static analysis. We show that automated tasks produced by our system work on 84% of configurations, while baseline automation techniques work on only 18% of configurations.

This paper has focused on use of our system for building an on-line repository of automated IT tasks which would include both local configuration and setup as well as remote tasks such as configuring a wireless router. We note, however, that our system is useful for many other applications as well, including replacing IT knowledgebases, automated software testing, and even use by expert users as an easy way to automate repetitive tasks.

Acknowledgments

We'd like to thank Steve Bauer and Neil Van Dyke for their help implementing an early version of the system, and Micah Brodsky and Martin Rinard for help with the mistake removal algorithm. Also, we greatly appreciate Hariharan Rahul's help editing an early draft of this paper, and Sam Perli and Nabeel Ahmed's help generating early results. Lastly we'd like to thank Regina Barzilay, S.R.K. Branavan, James Cowling, Evan Jones, Ramesh Chandra, Jue Wang, Carlo Curino, Lewis Girod and our shepherd Michael Isard for their feedback on the paper. This work was supported by NSF grant IIS-0835652.

References

[1] AutoHotkey. http://www.autohotkey.com/.
[2] AutoIt, a freeware Windows automation language. http://www.autoitscript.com/.
[3] Automator. http://developer.apple.com/macosx/automator.-html.
[4] eHow. http://www.ehow.com.
[5] IAcessibility2. http://www.linuxfoundation.org/en/Accessibility/-IAccessible2.
[6] KVM. http://www.linux-kvm.org.
[7] Mac OS X Accessibility Framework . http://developer.apple-.com/documentation/Accessibility/Conceptual/AccessibilityMac-OSX/AccessibilityMacOSX.pdf.
[8] Microsoft Active Accessibility. http://en.wikipedia.org/wiki/-Microsoft_Active_Accessibility.
[9] Microsoft Help. http://windows.microsoft.com/en-us/windows/help.
[10] Security Garden Blog. http://securitygarden.blogspot.com/2009/-04/microsoft-fix-it-gadget.html.
[11] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker. Netprints: Diagnosing home network misconfigurations using shared knowledge. In *NSDI*, 2009.
[12] M. Attariyan and J. Flinn. Using causality to diagnose configuration bugs. *USENIX*, 2008.
[13] T.-H. Chang, T. Yeh, and R. C. Miller. Gui testing using computer vision. In *CHI*, 2010.
[14] L. D. B. et. al. DocWizards: A System For Authoring Follow-me Documentation Wizards. In *UIST*, 2005.

E-mail	
Sending/ Receiving	Turn off E-mail Read Receipts (54, 27) Automatically forward e-mail to another address (35, 30)
Viewing	Restore the unread mail folder (16, 8) Highlight all messages sent only to me (31, 24) Change an e-mail filtering rule (18, 19) Add an e-mail filter rule (46, 26) Make the recipient column visible in the Inbox (27, 11) Order e-mail message by sender (19, 73) Create an Outlook Search Folder (12, 12) Turn on threaded message viewing in Outlook (16, 9) Mark all messages as read (44, 48) Automatically empty deleted items folder (22, 24)
Junkmail	Empty junk e-mail folder (9, 9) Turn off Junk e-mail filtering (22, 14)
Security	Consider people e-mailed to be safe senders (19, 25) Send an e-mail with a receipt request (20, 12)
Contacts/ Calendar	File Outlook contacts by last name (25, 13) Set Outlook to start in Calendar mode (15, 23)
RSS Feeds	Add a new RSS feed (14, 15) Change the Name of an RSS feed (12, 23)
Other	Turn off Outlook Desktop Alerts (24, 35) Reduce the size of a .pst file (26, 39) Turn off notification sound (22, 66) Switch calendar view to 24-hour clock (20, 14)
Office Applications	
Excel	Delete a worksheet in Excel (8, 8) Turn on AutoSave in Excel (33, 111)
Word	Disable add-ins in Word (25, 23)
Web	
Browser	Install Firefox (23, 21)
Proxy	Manually Configure IE SSL Proxy (61, 83) Set Default Http Proxy (7, 7)
Networking	
Security and Privacy	Enable firewall exceptions (9, 9) Enable Windows firewall (6, 6) Disable Windows firewall notifications (8, 9) Disable Windows firewall (6, 9)
IPv6	Disable IPv6 to IPv4 tunnel (8, 7) Show the current IPv4 routing table (10, 17) Show the current IPv6 routing table (13, 10)
DNS	Use OpenDNS (44, 38) Stop caching DNS replies (6, 9) Use Google's Public DNS servers (32, 32) Use DNS server from DHCP (22, 22)
Routing	Configure system to pick routes based on link speed (22, 17) Set routing interface metric (18, 19)
System	
Utilities	Analyze hard drive for errors (7, 13) Defragment hard drive (10, 13) Enable Automatic Updates (7, 6) Set Up Remote Desktop (12, 10)
User Interface Settings	Hide the Outlook icon in the System tray (21, 18) Change to Classic UI (15, 13) Delete an Item from the Task Bar (13, 9) Change desktop background color (35, 26) Enable Accessibility Options (20, 20) Auto-Hide the Taskbar (52, 41) Change date to Long Format (33, 19) Set Visual Effects for Performance (13, 13)
Other	Set Outlook as default E-mail program (26, 15) Enable Password on Screen Saver and Resume (22, 29)

Table 3: 57 tasks used to evaluate KarDo. Each task is listed with the number of actions performed in each of the two traces.

[15] Q. Huang, H. Wang, and N. Borisov. Privacy-Preserving Friends TroubleShooting Network. In *NDSS*, 2005.
[16] O. Laadan, R. A. Baratto, D. B. Phung, S. Potter, and J. Nieh. Dejaview: A personal virtual computer recorder. In *SOSP*, 2007.
[17] Olsen. *Developing User Interfaces*. Morgan Kaufmann, 1998.
[18] Y. Su, M.A., and J. F. Autobash: improving configuration management with operating system causality analysis. *SOSP*, 2007.
[19] Y.-Y. Su and J. Flinn. Automatically generating predicates and solutions for configuration troubleshooting. *USENIX*, 2009.
[20] H. J. Wang, J.P., Y.C., R.Z., and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI*, 2004.
[21] Y.-M. Wang and et. al. Strider: A black-box, state-based approach to change and configuration management and support. In *LISA*, 2003.

Automating configuration troubleshooting with dynamic information flow analysis

Mona Attariyan and Jason Flinn
University of Michigan

Abstract

Software misconfigurations are time-consuming and enormously frustrating to troubleshoot. In this paper, we show that dynamic information flow analysis helps solve these problems by pinpointing the root cause of configuration errors. We have built a tool called ConfAid that instruments application binaries to monitor the causal dependencies introduced through control and data flow as the program executes — ConfAid uses these dependencies to link the erroneous behavior to specific tokens in configuration files. Our results using ConfAid to solve misconfigurations in OpenSSH, Apache, and Postfix show that ConfAid identifies the source of the misconfiguration as the first or second most likely root cause for 18 out of 18 real-world configuration errors and for 55 out of 60 randomly generated errors. ConfAid runs in only a few minutes, making it an attractive alternative to manual debugging.

1 Introduction

Complex software systems are difficult to configure and manage. When problems inevitably arise, operators spend considerable time troubleshooting those problems by identifying root causes and correcting them. The cost of troubleshooting is substantial. Technical support contributes 17% of the total cost of ownership of today's desktop computers [24], and troubleshooting misconfigurations is a large part of technical support. For information systems, administrative expenses, made up almost entirely of people costs, represent 60–80% of the total cost of ownership [16]. Even for casual computer users, troubleshooting is often enormously frustrating.

In this paper, we show that system support for dynamic information flow analysis can substantially simplify and reduce the human effort needed to troubleshoot software systems. We focus specifically on configuration errors, in which the application code is correct, but the software has been installed, configured, or updated incorrectly so that it does not behave as desired. For instance, a mistake in a configuration file may lead software to crash, assert, or simply produce erroneous output.

Why address misconfigurations specifically? Empirical evidence exists that misconfigurations are often the dominant cause of problems in deployed systems. For example, Gray [20] attributed 42% of system outages to administration, while software, hardware, and environment failures account for 25%, 18%, and 14% of failures, respectively. Murphy and Gent [31] note that the percentage of failures attributable to system management is increasing over time, and that management failures have come to dominate the combination of software and hardware failures. Other studies have shown that configuration errors are the largest category of operator mistakes. Oppenheimer et al. [35] studied three commercial Internet services and found that more than 50% of the operator mistakes that led to service unavailability were misconfigurations. Nagaraja et al. [33] found that software misconfiguration was the most common type of operator mistake, accounting for more than half of all mistakes. Other studies have shown similar results [7, 8, 23]. Further, while fault tolerance techniques such as modular redundancy [30] or Byzantine fault tolerance [10] can mask software and hardware faults, they do not prevent human error such as an operator who misconfigures all replicas [20, 23].

Consider how users and administrators typically debug configuration problems. Misconfigurations are often exhibited by an application unexpectedly terminating or producing erroneous output. While an ideal application would always output a helpful error message when such events occur, it is unfortunately the case that such messages are often cryptic, misleading, or even non-existent. Thus, the person using the application must ask colleagues and search manuals, FAQs, and online forums to find potential solutions to the problem. Troubleshooting is a tedious, time-consuming process that can substan-

tially increase the time to recover (TTR) from a failure.

To remedy this problem, we have developed a tool, called ConfAid, that uses dynamic information flow analysis to identify the likely root cause of a configuration problem. When a user or administrator wishes to troubleshoot a problem such as a crash or incorrect output, she reproduces the problem while ConfAid modifies the executed application binaries to track the causal dependencies between configuration inputs and program behavior. ConfAid produces an ordered list of the configuration tokens most likely to have caused the exhibited problem. While dynamic analysis takes a few minutes for a complex application such as Apache, automated troubleshooting is still considerably faster and less labor-intensive than manual debugging or searching through FAQs and online forums.

ConfAid dynamically tracks causality (i.e., information flow) at a fine granularity, namely at the level of instructions and bytes. While there is a large body of work in the distributed systems community that tracks causality to understand and troubleshoot program behavior [2, 5, 6, 11, 12, 13], these prior systems essentially treat application binaries as black boxes, understanding causal relationships between processes by tracking network messages and IPCs. Some gain more information by inserting probes into applications to glean hints about their activity. ConfAid, however, "opens up the black-box" by examining the flow of causality *within* processes as they execute. Further, since ConfAid tracks causality using binary instrumentation [29], it does not require application source code to find misconfigurations.

ConfAid restricts the scope of information flow analysis to only track values that depend on data read from configuration files. ConfAid tracks dependencies introduced by both data and control flow. If it determines that altering a configuration parameter may change the application's control flow such that it avoids the problem (and does not exhibit a different problem), it reports that parameter as a possible root cause. It propagates dependencies among multiple processes in a distributed system by annotating IPCs and network communication.

Our results show that ConfAid identifies the correct root causes of most configuration errors. We injected 18 real-world misconfigurations into OpenSSH, Apache, and the Postfix email server. ConfAid identifies the correct root cause as the most likely source of the misconfiguration in 13 cases; for the remaining 5 bugs, it lists the correct root cause as the second most likely option. ConfAid analysis takes less than 3 minutes, making the tool an attractive alternative to manual troubleshooting.

2 Design principles

We next briefly describe ConfAid's design principles.

2.1 Use white-box analysis

The genesis of ConfAid arose from AutoBash [37], our prior work in configuration troubleshooting. AutoBash tracks causality at process and file granularity in order to diagnose configuration errors. It treats each process as a *black box*, such that all outputs of the process are considered to be dependent on all prior inputs. We found AutoBash to be very successful in identifying the root cause of problems, but the success was limited in that AutoBash would often identify a complex configuration file, such as Apache's `httpd.conf`, as the source of an error. When such files contain hundreds of options, the root cause identification of the entire file is often too nebulous to be of great use.

Our take-away lessons from AutoBash were: (1) causality tracking is an effective tool for identifying root causes, and (2) causality should be tracked at a finer granularity than an entire process to troubleshoot applications with complex configuration files. These observations led us to use a *white box* approach in ConfAid that tracks causality within each process at byte granularity.

The granularity of the root causes reported to the user is also much finer. Instead of reporting the entire configuration file as a root cause, ConfAid points its users to specific tokens in the configuration file that it believes to be in error. This approach narrows down root causes considerably for programs like Apache.

2.2 Operate on application binaries

We next considered whether ConfAid should require application source code for operation. While using source code would make analysis easier, source code is unavailable for many important applications, which would limit the applicability of our tool. Also, we felt it likely that we would have to choose a subset of programming languages to support, which would also limit the number of applications we could analyze.

For these reasons, we decided to design ConfAid to not require source code; ConfAid instead operates on program binaries. ConfAid uses Pin [29] to dynamically insert instrumentation into binaries as applications run. It also uses IDA Pro [22] to statically generate control flow graphs from binaries.

2.3 Embrace imprecise analysis

Our final design decision was to embrace an imprecise analysis of causality that relies on heuristics rather than using a sound or complete analysis of information flow. Using an early prototype of ConfAid, we found that for any reasonably complex configuration problem, a strict definition of causal dependencies led to our tool outputting almost all configuration values as the root cause

of the problem. Many registers and bytes in the address space would come to depend on almost all configuration values. Our prototype would identify the root cause as only one of many possible causes.

Thus, our current version of ConfAid uses several heuristics to limit the spread of causal dependencies. For instance, ConfAid does not consider all dependencies to be equal. It considers data flow dependencies to be more likely to lead to the root cause than control flow dependencies. It also considers control flow dependencies introduced closer to the error exhibition to be more likely to lead to the root cause than more distant ones. In some cases, ConfAid's heuristics can lead to false negatives and false positives. However, our results show that in most cases, they are quite effective in narrowing the search for the root cause and reducing execution time.

3 Design and implementation

3.1 Overview: How ConfAid runs

ConfAid is designed to be used by system administrators and end users when they encounter a suspected misconfiguration that they do not know how to fix. ConfAid is run offline, once erroneous behavior has been observed. A ConfAid user reproduces the problem by executing the application while ConfAid attaches to the executing application processes and monitors information flow within them. For non-deterministic bugs, ConfAid could potentially leverage one of several deterministic replay systems that can capture a buggy non-deterministic execution and faithfully reproduce it for later analysis [3, 18, 27, 36].

To use ConfAid, a user specifies: (1) which binaries ConfAid should monitor, (2) the sources of configuration data, and, as needed, (3) the erroneous external output of the application. For simple applications, ConfAid may monitor only a single process. For more complicated applications, ConfAid dynamically attaches to multiple specified processes and monitors inter-process dependencies as described in Section 3.5. While ConfAid could potentially monitor *any* process that receives input via IPC or a network message from a process already monitored by ConfAid, we decided to only monitor executables specified by the user in order to limit the scope of analysis. Our prior experience with AutoBash showed that many extraneous processes communicate with processes being debugged via channels such as files, pipes, and signals, yet these processes are not needed to determine the root cause.

Similarly, we could potentially treat *any* source of input to a program as a source of configuration data. However, such an approach would dramatically slow the analysis since most locations in the process address space would come to depend on one or more inputs. In contrast, ConfAid only monitors input from designated configuration sources. This makes ConfAid analysis more tractable than generic taint tracking or program slicing because the number of locations with dependencies is small. Typically, the sources to monitor are self-evident; e.g., `httpd.conf` is the configuration source for Apache. Potentially, we could automate this process by treating all inputs from specific locations (e.g., the `etc` directory) or files with semantic keywords (such as "`*.conf`") as configuration inputs.

Finally, a ConfAid user may designate specific error conditions. ConfAid automatically treats assertion failures and exits with non-zero return codes as an erroneous terminations. However, some misconfigurations lead not to program termination, but instead to the process producing erroneous output. We therefore allow the user to specify a particular string expression as erroneous. ConfAid monitors the system calls that write to network, terminal, and other external output channels. When it finds a matching output, it considers the output an error.

ConfAid outputs an ordered list of probable root causes. Each entry in the list is a token from a configuration source; our results show that ConfAid typically outputs the actual root cause as the first or second entry in the list. This allows the ConfAid user to focus on one or two specific configuration tokens when deciding how to fix the problem. By finding the needle in the haystack, ConfAid dramatically improves TTR.

3.2 Basic information flow analysis

In this section, we describe the basic information flow analysis used by ConfAid. For simplicity of explanation, we defer discussing optimizations and heuristics until Sections 3.3 and 3.4. We also assume that ConfAid is tracking only a single process; Section 3.5 describes how we extend ConfAid analysis to multiple cooperating processes on one or more computers.

ConfAid dynamically monitors the information flow from configuration sources through process memory and registers to the point in the program execution when erroneous behavior is observed. It does so by using Pin [29] to add custom logic, referred to as *instrumentation*, to the process binary. As described below, ConfAid instrumentation is executed before or after most x86 instructions executed by a monitored application.

ConfAid uses taint tracking [34] to analyze information flow. It inserts instrumentation into the binary that monitors each system call such as `read` or `pread` that could potentially read data from a configuration source. If the source of the data returned by a system call was specified as a configuration file, ConfAid annotates the registers and memory addresses modified by the system

```
if (c == 0) { /* c set to 0 in config file */
    x = a;    /* taken path */
} else {
    y = b;    /* alternate path */
}
z = d;
if (z) assert(); /* The erroneous behavior */
```

Figure 1: Example to illustrate causality tracking

call with a marker that indicates a dependency on a specific configuration token. Borrowing terminology from the taint tracking literature, we refer to this marking as the *taint* of the memory location. If an address or register is tainted by a token, ConfAid believes that the value at that location might be different if the value of the token in the original configuration source were to change.

We use the notation, T_x to denote the taint set of variable x. T_x is a set of configuration tokens; for instance, if $T_x = \{$ `FOO, BAR` $\}$, ConfAid believes that the value of variable x could change if the user were to modify either the `FOO` or `BAR` tokens in the configuration file. In the basic information flow analysis, taints are binary (a location is either tainted by a token or it is not); in Section 3.4, we attach a weight to each taint.

Taint is propagated via data flow and control flow dependencies. When a monitored process executes an instruction that modifies a memory address, register, or CPU flag, the taint set of each modified location is set to the union of the taint sets of the values read by the instruction. For example, given the instruction $x = y + z$ where the taint sets of y and z are T_y and T_z respectively, the taint set of x, T_x, becomes $T_y \cup T_z$. Intuitively, the value of x might change if a configuration token were to cause y or z to change prior to the execution of this instruction. For example, if $T_y = \{$ `FOO, BAR` $\}$ and $T_z = \{$ `FOO, BAZ` $\}$, then $T_x = \{$ `FOO, BAR, BAZ` $\}$.

In traditional taint tracking for security purposes, control flow dependencies are often ignored to improve performance because they are harder for an attacker to exploit. With ConfAid, however, we have found that tracking control flow dependencies is essential since they propagate the majority of configuration-derived taint.

A naive approach to tracking control flow is to union the taint set of a branch conditional with a running control flow dependency for the program. For example, on executing the statement `if (b)`, ConfAid could set the control flow taint set, T_{cf}, to $T_{cf} \cup T_b$. However, without mechanisms to *remove* taint from T_{cf}, control flow taint grows without limit. This causes too many false positives, i.e., ConfAid would identify most configuration tokens as possible root causes.

A more precise approach takes into account the basic block structure of a program. Consider the example in Figure 1. Assume a, b, c, and d were read from a configuration file and have taint sets T_a, T_b, T_c, and T_d, respectively (i.e., T_a is a set containing only configuration token a). The value of c does not affect whether the last two statements are executed, since they execute in all possible paths (and therefore for all values of c). Thus, T_c should be removed from T_{cf} before executing $z = d$. When the program asserts, T_{cf} should only include T_d in the example, to correctly indicate that changing the value of d might fix the problem.

ConfAid also tracks implicit control flow dependencies. In Figure 1, the values of x and y depend on c when the program asserts, since the occurrence of their assignments to a and b depend on whether or not the branch is taken. Note that y is still dependent on c even though the `else` path is not taken by the execution since the value of y might change if a configuration token is modified such that the condition evaluates differently.

When the program executes a branch with a tainted condition, ConfAid first determines the merge point (the point where the branch paths converge) by consulting the control flow graph. Prior to dynamic analysis, ConfAid obtains the graph by using IDA Pro to statically analyze the executable and any libraries it uses (e.g., `libc` and `libssl`).

For each tainted branch, ConfAid next explores each *alternate path* that leads to the merge point. We define an alternate path to be any path not taken by the actual program execution that starts at a conditional branch instruction for which the branch condition is tainted by one or more configuration values. ConfAid uses alternate path exploration to learn which variables would have been assigned had the condition evaluated differently due to a modified configuration value. The taint set of any variable assigned on an alternate path is set to the union of its previous taint set, the taint set of the conditional, and the taint set of the variables read by the assigning instruction. In the example, $T_y = T_y \cup T_c \cup \{T_c \wedge T_b\}$. In other words, a configuration token affecting the previous value of y could change, or c could change, causing the previous value of y to be overwritten. Finally, it might be necessary for both c and b to change (as denoted by the term $\{T_c \wedge T_b\}$) since c allows the alternate assignment, and b may need to reflect a correct configuration value.

To evaluate an alternate path, ConfAid executes the program by switching the condition outcome, similar to the predicate switching approach used by Zhang et al. [48] to explore implicit dependencies. ConfAid uses copy-on-write logging to checkpoint and roll back application state. When a memory address is first altered along an alternate path, ConfAid saves the previous value in an undo log. At the end of the execution, applica-

tion state is replaced with the previous values from the log. ConfAid uses Pin mechanisms to checkpoint and rollback the state of the processor, which includes the registers and CPU flags. Since some alternate paths are quite long, ConfAid uses a *bounded horizon heuristic* described in Section 3.3.1 to limit the number of instructions it explores along each alternate path. Many branches need not be explored since their conditions are not tainted by any configuration token.

After exploring the alternate paths, ConfAid performs a similar analysis for the path actually taken by the program. This is the actual execution, so no undo log is needed. In the example, analyzing the taken path would derive $T_x = T_a \cup T_c \cup \{T_c \wedge T_x\}$.

ConfAid also uses alternate path exploration to learn which paths avoid erroneous application behavior. ConfAid considers an alternate path to avoid the erroneous behavior if the path leads to a successful termination of the program or if the merge point of the branch occurs after the occurrence of the erroneous behavior in the program (as determined by the static control flow graph). ConfAid unions the taint sets of all conditions that led to such alternate paths to derive its final result. This result is the set of all configuration tokens which, if altered, could cause the program to avoid the erroneous behavior.

Figure 2 shows four examples that illustrate how ConfAid detects alternate paths that avoid the erroneous behavior. In case (a), the error occurs after the merge point of the conditional branch. ConfAid determines that the branch does not contribute to the error, because both paths lead to the same erroneous behavior. In case (b), the alternate path avoids the erroneous behavior because the merge point occurs after the error, and the alternate path itself does not exhibit any other error. In this case, ConfAid considers tokens in the taint set of the branch condition as possible root causes of the error, since if the application had taken the alternate path, it could have avoided the error. In case (c), the alternate path leads to a different error (an assertion). Therefore, ConfAid does not consider the taint of the branch as a possible root cause because the alternate path would not lead to a successful termination. In case (d), there are two alternate paths, one of which leads to an assertion and one that reaches the merge point. In this case, since there exists an alternate path that avoids the erroneous behavior, configuration tokens in the taint set of the branch condition are possible root causes.

One limitation of evaluating an alternate path with predicate switching is that switching a predicate outcome, but not the underlying data values, may result in an "unnatural" execution that leads to erroneous behaviors, such as a crash due to a segmentation fault. In such circumstances, ConfAid aborts exploration of the alternate path but conservatively retains the taint of the conditional

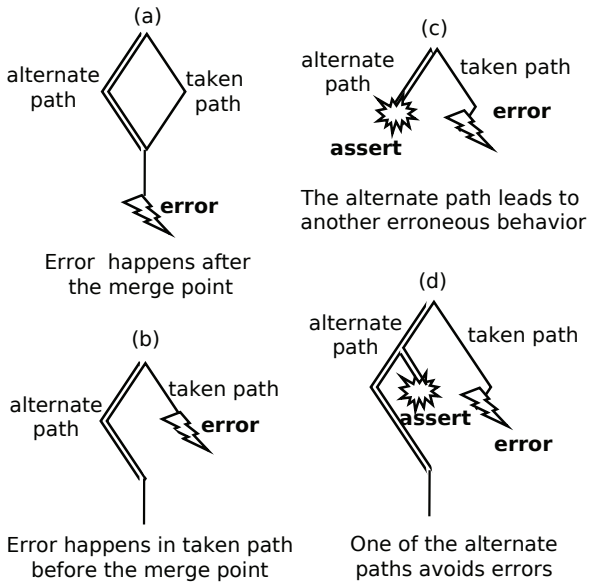


Figure 2: Examples illustrating ConfAid path analysis

branch in the possible root causes. This conservative behavior may lead to false positives if the alternate path would in fact lead to a real error later in the execution. The early abort of the alternate path may also lead to false negatives due to unexplored variable assignments.

3.2.1 Abstracting library functions and system calls

There are three cases where ConfAid does not dynamically analyze information flow. The first case is when the application makes a system call. Since ConfAid does not track taint inside the operating system, the information flow analysis stops at the system call entry. The second case is commonly executed standard library functions such as `malloc` in `libc` and cryptographic functions in `libssl`. ConfAid uses a primitive static analysis for these functions to improve analysis speed while still producing the identical effect on process taint values that would have been produced by a fully-instrumented execution. Since we abstract only functions in standard libraries, such taint abstractions are application-independent. The final case is a small number of heavily optimized `libc` functions for which IDA Pro does not produce a complete static analysis.

To handle these cases, ConfAid uses *taint abstraction* of the function (or system call). A taint abstraction specifies how taint is propagated from the inputs of the functions to its outputs (e.g., return values and modified location in the address space). When a process calls one of these functions, ConfAid first executes the function

without any instrumentation and then uses the taint abstraction to modify the taints of the process memory and registers.

3.3 Heuristics for performance

ConfAid uses two heuristics to simplify control flow analysis. These heuristics eliminate exploration of some alternate paths to concentrate on the paths that are most likely to be useful in identifying the root cause. The heuristics reduce analysis time but also introduce false positives and negatives.

3.3.1 The bounded horizon heuristic

The first heuristic is the *bounded horizon* heuristic. ConfAid only executes each alternate path for a fixed number of instructions. By default, ConfAid uses a limit of 80 instructions. All addresses and registers modified within the limit are used to calculate information flow dependencies after the merge point. Locations modified after the limit do not affect dependencies introduced at the merge point. If an alternate path contains further tainted conditional branches, ConfAid executes each path until the limit is reached. For example, if the limit is 80 instructions and a tainted conditional branch occurs after executing 50 instructions, both paths from the new branch are executed for an additional 30 instructions.

3.3.2 The single mistake heuristic

The second heuristic simplifies control flow analysis by assuming that the configuration file contains only a limited number of erroneous tokens. By default, ConfAid assumes that the configuration file contains a single error — we refer to this as the *single mistake* heuristic.

To illustrate how this simplifies path exploration, consider again the example in Figure 1. Recall that at the time the assert statement is executed, $T_x = T_a \cup T_c \cup \{T_c \wedge T_x\}$. The single mistake heuristic eliminates the last term since that term requires the values of two tokens to change simultaneously. Similarly, ConfAid derives $T_y = T_y \cup T_c$ during alternate path exploration. Note that T_y no longer depends upon T_b. This seems counterintuitive, but for the assignment $y = b$ to occur in the program, a token in T_c must change to cause the alternate path to be taken. With the single mistake heuristic, a token in T_b but not in T_c cannot be the root cause, since one token in T_c already must change.

More importantly, restricting the number of configuration values that can change reduces the alternate paths that are explored, as shown in Figure 3. The nested condition, $c2$, can change only if a single configuration value affects both $c1$ and $c2$. If $T_{c1} \cap T_{c2} = \emptyset$, then the alternate path of $c2$ need not be explored at all.

```
if (c1 == 0) { /* c1 set to 0 in config file */
    ...
} else {
    if (c2 == 0) { /* c2 set to 0 also */
        x = a;
    } else {
        y = b;
    }
}
```

Figure 3: Example to illustrate alternate path pruning

To implement this heuristic, we introduce a new variable, T_{alt}, that is the set of configuration options that, if changed, would cause the execution of the program to reach the current instruction. Initially, T_{alt} is the set of all configuration tokens. At each condition, c, T_{alt} does not change along the taken path, but we set $T_{alt} = T_{alt} \cap T_c$ along the alternate path. In Figure 3, $T_{alt} = T_{c1} \cap T_{c2}$ after the second condition. When T_{alt} is $\emptyset$, the alternate path is explored no further. When a variable is assigned along an alternate path, its taint value is set to the union of its previous taint set and T_{alt}. Thus, $T_x = T_x \cup T_{c1}$ and $T_y = T_y \cup (T_{c1} \cap T_{c2})$.

The single mistake heuristic may lead to false negatives. In Figure 3, if $c1$ and $c2$ are tainted by a disjoint set of tokens, ConfAid will not explore the path on which y is assigned to b, so it may miss the root cause if the program later asserts based on the value of y. Potentially, if ConfAid cannot find a root cause, we can relax the single-mistake assumption by allowing ConfAid to assume that two or more tokens are erroneous. In our experiments to date, this heuristic has yet to trigger a false negative.

3.4 Heuristics for reducing false positives

We originally designed ConfAid to use only the basic taint tracking algorithm described in Section 3.2 with the bounded horizon and single mistake heuristics. However, our initial experiments with this design met with only limited success. Typically, ConfAid would include the root cause of a misconfiguration in its output set, yet the cardinality of the output set would be very large. For many bugs, ConfAid would return a significant fraction of the tokens in the configuration file.

In analyzing our initial results, we realized that it was insufficient to track information flow dependencies as binary values. In our design as described so far, two configuration tokens are considered equal taint sources even if one has a direct causal relationship to a location (e.g., the value in memory was read directly from the configuration file) and another has a nebulous relationship (e.g.,

the taint was propagated along a long chain of conditional assignments deep along alternate paths).

Another problem we noticed was that loops could cause a location to become a global source and sink for taint. For instance, Apache reads its configuration values into a linked list structure, and then traverses the list in a loop to find the value of a particular configuration token. During the traversal, the program control flow picks up taint from many configuration options, and these taints are sometimes transferred to the configuration variable that is the target of the search.

We realized that both of these problems were caused by the implicit assumption in our design that all information flow relationships should be treated equally. Essentially, our design had no shades of gray: it either considered a location to be tainted by a token or it did not. Based on this observation, we decided to modify our design to instead track taint as a floating-point weight ranging in value between zero and one. For example, the taint of x might be represented as { FOO:w_{foo}, BAR:w_{bar} }. As before, this set indicates that modifying either token FOO or BAR might change the value of x. However, if $w_{foo} > w_{bar}$, FOO has a more direct relationship to x, and hence is believed to be a better candidate for the root cause of an error that depends on x.

We revised ConfAid to use heuristics to weight the dependencies introduced by information flow differently, with those relationships that are more likely to lead to the root cause given a higher weight than those less likely to lead to the root cause. We also modified ConfAid to order the set of tokens on which an erroneous behavior depends by their respective weights before outputting them.

Our weights are based on two heuristics. First, data flow dependencies are assumed to be more likely to lead to the root cause than control flow dependencies. Second, control flow dependencies are assumed to be more likely to lead to the root cause if they occur later in the execution (i.e., closer to the erroneous behavior).

Specifically, we assign taints introduced by control flow dependencies only half the weight of taints introduced by data flow dependencies. Further, each nested conditional branch reduces the weight of dependencies introduced by prior branches in the nest by one half. We chose a weight of 0.5 for speed: it can be implemented efficiently with a vector bit shift.

For example, in Figure 4, the assignment $x = a$ is a data flow dependency, so $T_x = T_a$ (any dependencies from a are inherited at full weight). However, y inherits taint from $c1$ through a control flow dependency. Thus, $T_y = max(T_a, \frac{T_{c1}}{2})$. That is, we weight any taint from $c1$ by half, while taint inherited from a is given full weight. We use a special *max* operator here rather than a simple union operator, since the values are now floating point rather than binary. Specifically, $max(T_x, T_y)$ produces a

```
x = a;
if (c1 == 0) { /* c1 set to 0 in config file */
    y = a;
} else {
    z = b;
}
if (c2 == 0) { /* c2 set to 0 in config file */
  if (c3 == 0) { /* c3 also set to 0 */
     w = a;
  }
}
```

Figure 4: Example to illustrate the weighting heuristic

set that contains all tokens that occur in either T_x and T_y. If a token appears in only one of T_x or T_y, its weight is set to its weight in that set. If a token appears in both T_x and T_y, its weight is set to the maximum of its weight in either set.

Similarly, $T_z = max(T_z, \frac{T_{c1}}{2})$ (recall that with binary values, $T_z = T_z \cup T_{c1}$ due to the single mistake heuristic). When ConfAid explores an alternate path, it replaces the intersection operator with a corresponding *min* operator. Thus, in the prior example from Figure 3, $T_y = max(T_y, min(\frac{T_{c1}}{4}, \frac{T_{c2}}{2}))$.

Figure 4 also shows two nested conditions. In calculating the taint of w, condition $c3$ is considered more influential than condition $c2$ because it occurs later in the program execution. Therefore $T_w = max(T_a, \frac{T_{c3}}{2}, \frac{T_{c2}}{4})$. The same weighting applies to alternate path execution; assignments on an alternate path starting at the $c3$ branch are given twice the weight as those on an alternate path starting at the $c2$ branch.

ConfAid also weights alternate paths that avoid the erroneous behavior by their proximity to the point in application execution where the behavior is exhibited. Paths starting from the closest tainted conditional branch that avoids the erroneous behavior are given full weight, those from the next closest branch are given half weight, and so on. Note that if a configuration token has a much stronger weight on the condition of a distant branch than any tokens for closer branches, ConfAid may still rank it as the most likely root cause.

Of course, when programs do not behave as expected, ConfAid's heuristics may lead to incorrect results. For example, an application could potentially execute a substantial amount of code between the point where the erroneous behavior occurs and the point where the program outputs some value that exhibits the error (e.g., an error message). If that code contains a condition tainted by a configuration token other than the one that caused the error *and* that condition changes the specific error message that is generated, ConfAid might identify the wrong token as the most likely root cause. While such a sce-

nario is uncommon, we did observe a single Apache bug (described further in the evaluation) in which ConfAid's heuristic failed in this manner.

3.5 Multi-process causality tracking

The most difficult configuration errors to troubleshoot involve multiple interacting processes. Such processes may be on a single computer, or they may reside on multiple computers connected by a network. To troubleshoot such cases, ConfAid instruments multiple processes at the same time and propagates taint information along with the data sent when the processes communicate.

ConfAid supports processes that communicate using sockets and files. The socket support includes Unix sockets and pipes, as well as UDP and TCP sockets. ConfAid instruments the system calls that create sockets and pipes. It marks these objects as taint propagating channels if the destination is another instrumented process. Then, ConfAid intercepts all sends and receives using those channels. When data is sent, ConfAid appends a header that indicates whether or not the data is tainted and, when applicable, the exact taint of the data. Taint information is propagated at per-byte granularity if the taints of different bytes of the buffer are different. On the receiving side, ConfAid extracts the header from the received data and assigns the indicated taints to the received data.

For files, ConfAid creates an auxiliary file with a special ".confaid" extension when an instrumented process writes tainted data to a file. The auxiliary file records which bytes in the corresponding file are tainted and the specific values of those taints. Like sockets, file taint is recorded at granularities as small as one byte. For instance, the file "foo.confaid" records the tainted bytes in file "foo". When an instrumented process reads data from a file and a corresponding auxiliary file exists, ConfAid sets the taints of bytes read from the file to the values specified in the auxiliary file.

Since these operations are performed by PIN instrumentation immediately before and after system call execution, the taint propagation is hidden from the application. No operating system modifications are needed.

3.6 Limitations and future work

Since configuration troubleshooting is complex, ConfAid makes a number of assumptions to simplify its analysis. First, ConfAid only troubleshoots configuration problems that lead to crashes, assertion failures, and incorrect output; it does not yet help diagnose misconfigurations that cause poor performance. One approach to tackling performance problems that we are investigating is to first use statistical sampling to associate use of a bottleneck resource such as disk or CPU with specific points in the program execution. Then, ConfAid-style analysis can determine which configuration tokens most directly affect the frequency of execution of those points.

Second, like previous configuration troubleshooting systems [38, 39], ConfAid currently assumes that the configuration file contains only one erroneous token. If fixing a particular error requires changing two tokens, then ConfAid's alternate path analysis may not identify both tokens, as described in Section 3.3.2. However, if a file contains two incorrect tokens that represent independent mistakes, ConfAid can tackle the two errors sequentially by first identifying the token that leads to the most immediate failure, and then identifying the other token once the first error is corrected. The single mistake heuristic improves ConfAid's performance by reducing the set of possible taints tracked during dynamic analysis. In the future, we plan to allow ConfAid to track sets of two or more misconfigured tokens and measure the resulting performance overhead. Potentially, we may use an expanding search technique in which ConfAid initially performs an analysis assuming only a single mistake, and then performs a lengthier analysis allowing multiple mistakes if the first analysis does not yield satisfactory results.

4 Evaluation

Our evaluation answers the following questions:

- How effective is ConfAid in identifying the root cause of configuration problems?
- How long does ConfAid take to find the root cause?

4.1 Experimental setup

We evaluated ConfAid on three applications: the OpenSSH server version 5.1, the Apache HTTP server version 2.2.14, and the Postfix mail transfer agent version 2.7. All of our experiments were run on a Dell OptiPlex 980 desktop computer with an Intel Core i5 Dual Core processor and 4 GB of memory. The machine runs Linux kernel version 2.6.21. For Apache, ConfAid instruments a single process; for OpenSSH and Postfix, multiple processes are instrumented.

To evaluate ConfAid, we manually injected errors into correct configuration files. Then, we ran a test case that caused the error we injected to be exhibited. We used ConfAid to instrument the process (or processes) for that application, and obtained the ordered list of root causes found by ConfAid. We use two metrics to evaluate ConfAid's effectiveness: the ranking of the actual root cause,

i.e., the injected mistake, in the list returned by ConfAid and the time to execute the instrumented application.

We used two different methods to generate configuration errors. First, we injected 18 real-world configuration errors that were reported in online forums, FAQ pages, and application documentation. Second, we used the ConfErr tool [25] to inject random errors into the configuration files of the three applications.

4.2 Real-world misconfigurations

We searched forums, FAQ pages and configuration documents to find actual configuration problems that users have experienced with our target applications. In total, we chose 18 misconfigurations (5–7 for each application) that were caused by errors in the configuration files. The 18 misconfigured values cover a range of data types, such as binary options, enumerated types, numerical ranges, and text entries such as server names. Table 1 lists the configuration errors for each application, as well as the ConfAid results.

In these experiments, ConfAid tracks dependencies among multiple processes for all OpenSSH and Postfix bugs. For OpenSSH, it instruments two processes that communicate via Unix sockets. For Postfix, it instruments between four and six processes that communicate via Unix sockets and files; the number of instrumented processes varies depending on how many processes are started before a particular bug manifests. Multi-process causality tracking is necessary to diagnose 4 out of 5 Postfix and 3 out of 7 OpenSSH bugs. For Apache, ConfAid does not track dependencies across processes since that application starts only a single process.

As shown in Table 1, ConfAid is highly effective in pinpointing the root cause of misconfigurations. ConfAid ranks the actual root cause first in 13 cases, and second in the other 5. Sometimes, when the actual root cause is ranked second, the token ranked first provides a valuable clue to help debug the problem. For instance, in Apache the actual error usually occurs nested inside a section or directive command in the config file. For the two Apache errors where the root cause is ranked second, the top-ranked option is the section or directive containing the error.

The performance of ConfAid is reasonable. The time to manifest the buggy behavior varies among applications. Postfix and OpenSSH take less than 2 minutes, while Apache takes 2–3 minutes to complete. The average execution time of 1:32 minutes is much faster and less frustrating than trying to fix such configuration errors by looking at the logs, searching the Internet, and asking colleagues for potential clues. For instance, the 6th Apache misconfiguration in Table 1 is taken from a thread in linuxforums.org [28]. After trying to fix the misconfiguration for quite a while, the user went to the trouble of posting the question in the forum and waited two days for an answer. In contrast, ConfAid identified the root cause in less than 3 minutes.

4.3 Effect of the weighting heuristic

We next examine the effect of the weighting heuristic introduced in Section 3.4. For each of the 18 real-world misconfigurations, we disabled the heuristic and re-ran ConfAid. With the heuristic disabled, ConfAid treats all sources of information flow equally. Therefore, instead of producing a ranked list of possible root causes, ConfAid returns a single set of tokens, each of which is considered equally likely to be the root cause.

The last column of Table 1 shows the number of false positives when the heuristic is disabled. In every case, ConfAid identifies the correct root cause as one of the returned tokens. However, the number of other tokens returned varies substantially. Without the heuristic, there were only two misconfigurations (the 6th OpenSSH bug and the 5th Postfix bug) for which ConfAid produces no false positives. For six other bugs, the number of false positives is relatively low (less than 6). For the remaining 10 bugs, ConfAid returns almost all options as possible root causes. Thus, without the weighting heuristic, ConfAid is ineffective for 55% of the misconfigurations.

4.4 Effects of bounded horizon heuristic

We next investigated the effect of varying ConfAid's limit on the number of instructions executed along each alternate path (discussed in Section 3.3.1) from the default value of 80 instructions. As Figure 5 shows, varying the limit has substantially different effects on execution time, depending on the application being instrumented. For OpenSSH (bug #1), the execution time increases approximately linearly from 56 seconds with no alternate path exploration to 2:29 minutes with a horizon of 1600 instructions. On the other hand, Postfix (bug #1), shows an apparently exponential growth as the bound increases. The execution time starts at 21 seconds with no alternate path exploration and increases to 7:10 minutes for a horizon of 800 instructions. With a horizon of 1600, ConfAid analysis did not complete.

This difference in behavior derives from the nature of the applications. We found that even with a limit of 80 instructions, more than 80% of the tainted conditional branches in the OpenSSH bug reach their merge points for all alternate paths. Increasing the horizon only affects a small fraction of the branches since the rest are short enough to finish within the limit. On the other hand, for Postfix, less than 50% of the branches reach their merge point within the limit of 80 instructions. As we raise the

Application	Bug	Description of misconfiguration	Total # of options	ConfAid rank of the root cause	Execution time	# false positives w/o weights
OpenSSH Server	1	The PermitRootLogin option is disabled. Therefore, the user cannot ssh as root. The server keeps denying permission although the password is entered correctly.	47	2^{nd} (tied w/1)	1m 16s	6
	2	The server only has the PasswordAuthentication option enabled, while the user can only authenticate via RSA keys.	47	1^{st} (tied w/1)	1m 10s	1
	3	The user does not have his public key in the directory specified in the SSH server config file. Therefore, he cannot authenticate.	48	2^{nd}	51s	43
	4	The user is not in the AllowUsers list in the SSH config file. Therefore, he cannot connect to the server although he enters the password correctly.	49	2^{nd}	48s	44
	5	The MaxAuthTries option in SSH server config is set too low. Therefore, the user is disconnected if she enters her password incorrectly once.	47	1^{st}	1m 13s	43
	6	The MaxStartups options is set to 1. Therefore, the server refuses to start a new session, while another unauthenticated session is still in progress.	47	1^{st}	9s	0
	7	The location of the server RSA key is not set correctly in the config file. Therefore, the client fails to verify the host key.	47	1^{st} (tied w/1)	36s	43
Apache HTTP Server	1	The path specified in the DocumentRoot option does not have a corresponding <Directory> section. Therefore, all accesses to this path are denied according to the default policy.	88	2^{nd} (tied w/1)	2m 46s	87
	2	The cgi-bin directory is ScriptAlised in the config file. This prevents the DirectoryIndex from working as expected. Therefore, the user cannot access the index file in the directory.	89	1^{st}	2m 45s	87
	3	The cgi-bin directory is aliased in the config file. However, the corresponding Directory section does not provide sufficient permissions. Therefore, accesses to this directory are denied.	89	2^{nd} (tied w/1)	2m 45s	88
	4	A virtual host with the same interface coverage is set for the HTTP server. This host points to a different DocumentRoot which overwrites the default one. Therefore, the user gets an index file with incorrect content upon accessing the server DocumentRoot.	93	1^{st}	2m 59s	91
	5	The cgi-bin directory is aliased and a CGI Handler is activated in the config file. However, the corresponding <Directory> section does not have the ExecCGI option set. The user cannot access the executables in this directory.	89	1^{st}	2m 46s	88
	6	A specific directory in DocumentRoot is also aliased to another directory outside DocumentRoot. Therefore, accesses to files in the first directory are redirected to the aliased directory, and the files are not found.	89	1^{st} (tied w/1)	2m 47s	86
Postfix	1	The mydestination option is not set correctly in the Postfix config file. Therefore, Postfix cannot deliver mail locally.	27	1^{st}	37s	4
	2	The myorigin option is set incorrectly in the Postfix config file. Therefore, the next relay host bounces the mail sent from the user's machine to the Internet.	27	1^{st}	1m 10s	4
	3	The relayhost option is set incorrectly. Therefore, Postfix cannot forward the email sent from the user's machine to the Internet.	29	1^{st}	47s	4
	4	The type of alias_maps option is not supported in the user's machine. Therefore, Postfix fails to send any mail locally or to the Internet.	29	1^{st}	32s	2
	5	The email address provided in luser-replay is not reachable. Therefore, Postfix cannot redirect other mail with wrong recipient to the luser-replay.	29	1^{st}	1m 38s	0

Table 1: Results for 18 real-world configuration bugs

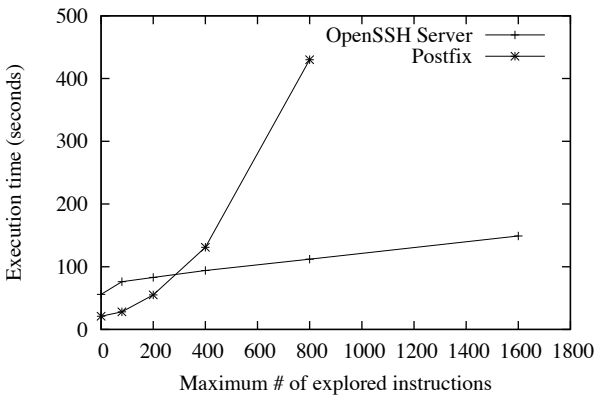


Figure 5: The effect of varying the horizon

limit, the percentage of the completed branches increases slowly to 60%.

To summarize, we found that there is no single limit that works best for all applications. Consequently, we envision that we could augment ConfAid to use an iterative search process in which it would start with a small horizon to generate results quickly, then continue to execute with larger horizons to refine the results.

4.5 Random fault injection

We next used ConfErr [25] to randomly generate configuration errors. ConfErr uses human error models rooted in psychology and linguistics to generate realistic configuration mistakes. We used ConfErr to produce 20 errors for each application. We then injected the errors one by one and measured the effectiveness and performance of ConfAid.

As shown in Table 2, ConfAid performs very well on these errors. The average time to execute all three applications is lower than the average execution time for the real-world errors used in the previous section. The main reason for this difference is that the real-world errors are often more complex than the randomly-generated ones. Therefore, it takes more time for the application to manifest the buggy behavior for real-world errors.

For the randomly generated errors, ConfAid instruments up to two processes for OpenSSH and up to six processes for Postfix. However, many faults are exhibited before these applications start additional processes; in such cases, ConfAid only instruments one process.

For OpenSSH, ConfAid successfully pinpointed the root cause (where we define success as listing the actual root cause as one of the top two options) for 95% of the bugs. For the last bug, ConfAid could not run to completion due to unsupported system calls used in the code path. We could remedy this by abstracting more calls.

ConfAid also successfully diagnoses 95% of the Apache errors. For the remaining error, ConfAid ranks the root cause 9th. The configuration error is that the DirectoryIndex file for the main document root is listed incorrectly in the Apache configuration file. The DirectoryIndex file is the file that Apache serves if that directory is accessed without mentioning a specific file. For instance, accessing `http://server.com/images/` will return the DirectoryIndex file listed for the `images` directory. However, the `Indexes` option is also activated for the document root directory. This option allows Apache to send the list of the files in the directory if no specific file in that directory is requested. The combination of these two options causes Apache to serve the list of files in the main document directory instead of the index file. ConfAid determines that the content sent to the user is dependent on the `Indexes` and related options first and the DirectoryIndex option next. Thus, the root cause gets ranked lower in the list. This ordering is a direct result of the heuristic discussed in Section 3.4 that considers branches closer to the erroneous behavior to be more likely to lead to the root cause than those farther away.

For Postfix, ConfAid diagnoses 85% of the errors effectively. The remaining 3 errors are due to missing configuration options. Currently, ConfAid only considers all tokens present in the configuration file as possible sources of the root cause. If a default value can be overridden by a token not actually in the file, then ConfAid will not detect the missing token as a possible root cause. Based on these results, we plan to extend our alternate path analysis to look for tokens that could be read from the config file along branches that are not actually executed. We can taint variables modified along those branches with a value that is dependent upon the branch conditions that led to that path.

Overall, ConfAid successfully diagnosed 55 out of 60 random errors by ranking the actual root cause first or second. Out of the remaining 5 errors, we believe that 4 (the OpenSSH server error and the three Postfix errors) can be diagnosed with further improvements to the ConfAid implementation. The remaining error (the Apache error) is a direct result of our weighting heuristic and seems hard for ConfAid to diagnose correctly.

5 Related work

Several prior research efforts have applied different techniques to the problem of configuration troubleshooting. Unlike ConfAid, most prior systems have taken a *black box* approach that uses only state external to the application being debugged to infer the problem.

PeerPressure [38] and its predecessor, Strider [39], use statistical methods to compare configuration state in the Windows registry on different machines. When a value

Application	root causes ranked first	root causes ranked first with one tie	root causes ranked second	root causes ranked second with one tie	root causes ranked worse than second	Avg. time to run
OpenSSH	17 (85%)	1 (5%)	1 (5%)	0	1 (5%)	7s
Apache	17 (85%)	1 (5%)	0	1 (5%)	1 (5%)	24s
Postfix	15 (75%)	0	2 (10%)	0	3 (15%)	38s

Table 2: Random fault injection results

on a machine exhibiting erroneous behavior differs from the value usually chosen by other machines, PeerPressure flags the value as a potential error. This approach works well as long as the majority configuration is appropriate for the target machine; however, PeerPressure and Strider cannot separate custom configuration variables from erroneous ones since they do not observe how applications actually use those values. In contrast, ConfAid can differentiate these cases by observing how the values are used inside the application binary.

Chronus [40] also compares multiple configuration states, Instead of comparing states across computers, it uses virtual machine checkpoint and rollback to "time travel" through states on the same machine, looking for the instance in which the program behavior on a particular test case switched from correct to incorrect.

Other projects monitor state external to applications as they run. Cohen et al. [15] use statistical techniques to help troubleshoot performance issues by correlating those issues with low-level performance statistics for the CPU, disk, and other system components. AutoBash [37] traces causality inside the OS by monitoring system call execution, but treats execution inside each process as a black box. AutoBash can suggest that a particular configuration file may be erroneous, but it cannot identify the specific value within the file that is at fault.

Our previous work on misconfiguration diagnosis [4] uses the application's system call trace to extract the files and processes on which the application causally depends. It then generates a signature based on those dependencies to represent the misconfiguration and search for the signature in a database of known bugs. Clarify [21] uses similar execution signatures to improve error reporting. It uses program features such as function call counts, call sites, and stack dumps to generate the signatures. The improved error reporting, although helpful, does not diagnose the root cause.

In contrast to all these projects, ConfAid takes a *white box* approach to configuration troubleshooting by monitoring causality within the program binary as it executes. Thus, ConfAid can observe the actual dependencies as they are introduced rather than inferring them through statistical and other methods.

Two recent systems apply white box analysis to a related problem: helping developers replicate a problem experienced in the field. SherLog [43] and ESD [44] both use static analysis and symbolic execution to infer the execution path of the application. SherLog uses log messages and ESD leverages the bug report generated by the application to constrain the execution path. Both of these systems can replicate an execution path that derives from a misconfiguration. However, they make different design decisions than ConfAid, driven by their different use case. They both require application source code, and SherLog also may require guidance from developers about which functions should be symbolically executed. This is appropriate for a tool used by software experts, but less so for one like ConfAid that is targeted at administrators and users. More generally, symbolic execution systems have been applied to model checking file systems and other complex software systems [9, 41].

A number of systems trace causality external to processes to debug configuration and performance issues in distributed systems. For example, the work of Aguilera et al. [2] and Magpie [5, 6] trace RPCs and other network communication to debug performance problems. Causeway [11] allows applications to inject metadata that follows causal paths for distributed applications. Pinpoint [13] traces middleware and communications between components in a distributed system and statistically correlates traces with success and failure data. Follow-on work to Pinpoint [12] adds the abstraction of causal *paths* that link black-box components. ConfAid and these systems share the common idea of propagating causal information among distributed components; however, ConfAid also propagates causal information within processes, which allows it to precisely determine the causal relationships between inputs and outputs.

More generally, many systems reason about causal interactions in the operating system and in distributed systems. For example, taint tracking [34] monitors data flow dependencies to determine when input data is used in an insecure manner. ConfAid uses the same approach for data flow analysis, but applies it to a different domain. Dytan [14] proposes a generic dynamic taint analysis framework to ease the implementation of various taint-based techniques. ConfAid enhances the basic dynamic taint analysis with essential heuristics and applies it to configuration troubleshooting problem. RedFlag [17] uses data flow analysis to reduce the leaks of

sensitive information by personal machines. Resin [42] uses application-level data flow assertions to improve the security of applications. Decentralized information flow [32, 45] monitors both control flow and data flow dependencies to determine if a code component leaks information that it is not authorized to divulge. BackTracker [26] traces causal interactions to determine what state has been changed during an intrusion. Asbestos [19] and HiStar [46] monitor causality in the OS to prevent inadvertent disclosure of private data.

Program slicing [1, 48, 47], intended to aid in debugging, is a more general approach that determines which statements could affect the value of a variable using a backward or forward computations. ConfAid applies similar data and control flow analysis techniques to a new problem, namely determining the root causes of misconfigurations.

6 Conclusion

Configuration errors are costly, time-consuming, and frustrating to troubleshoot. ConfAid makes troubleshooting easier by pinpointing the specific token in a configuration file that led to an erroneous behavior. Compared to prior approaches, ConfAid distinguishes itself by analyzing causality *within* processes as they execute without the need for application source code. It propagates causal dependencies among multiple processes and outputs a ranked list of probable root causes. Our results show that ConfAid usually lists the actual root cause as the first or second entry in this list. Thus, ConfAid can substantially reduce total time to recovery and perhaps make configuration problems a little less frustrating.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Shan Lu, for comments that improved this paper. We also thank the ConfErr team for helping us use their tool. This research was supported by NSF award CNS-1017148. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the University of Michigan, or the U.S. government.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[2] M. K. Aguilera, J. C. Mogul, J. L. wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 74–89, Bolton Landing, NY, October 2003.

[3] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 193–206, October 2009.

[4] M. Attariyan and J. Flinn. Using causality to diagnose configuration bugs. In *Proceedings of the USENIX Annual Technical Conference*, pages 171–177, Boston, MA, June 2008.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 259–272, San Francisco, CA, December 2004.

[6] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, May 2003.

[7] A. B. Brown and D. A. Patterson. To err is human. In *DSN Workshop on Evaluating and Architecting System Dependability*, Goteborg, Sweden, July 2001.

[8] A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the 2003 USENIX Technical Conference*, San Antonio, TX, June 2003.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Usenix Symposium on Operating System Design and Implementation (OSDI)*, pages 209–224, December 2008.

[10] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[11] A. Chanda, K. Elmeleegy, A. L. Cox, and W. Zwaenepoel. Causeway: Operating system support for controlling and analyzing the execution of distributed programs. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)*, Santa Fe, NM, June 2005.

[12] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

[13] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem, determination in large, dynamic Internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 595–604, Bethesda, MD, June 2002.

[14] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *In Proceedings of the International Symposium on Software Testing and Analysis*, pages 196–206, July 2007.

[15] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 231–244, San Francisco, CA, December 2004.

[16] Computing Research Association. Final report of the CRA conference on grand research challenges in information systems. Technical report, September 2003.

[17] L. P. Cox and P. Gilbert. RedFlag: Reducing inadvertant leaks by personal machines. Technical Report MSR-TR-2009-02, Duke University, 2009.

[18] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Boston, MA, December 2002.

[19] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris.

Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, United Kingdom, October 2005.

[20] J. Gray. Why do computer stop and what can be done about it? Technical Report 85.7, Tandem Corp., June 1985.

[21] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black-box components. In *Proceedings of the Conference on Programming Language Design and Implementation 2007*, pages 101–111, San Diego, CA, 2007.

[22] IDA Pro disassembler. http://www.hex-rays.com/idapro.

[23] F. Junqueira, Y. J. Song, and B. Reed. BFT for the skeptics. In *ACM Symposium on Operating Systems Principles: Work in Progress Session*, October 2009.

[24] A. Kapoor. Web-to-host: Reducing total cost of ownership. Technical Report 200503, The Tolly Group, May 2000.

[25] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 157–166, Anchorage, AK, June 2008.

[26] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 223–236, Bolton Landing, NY, October 2003.

[27] D. Lee, B. Wester, K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 77–89, Pittsburgh, PA, March 2010.

[28] http://www.linuxforums.org/forum/servers/125833-solved-apache-wont-follow-symlinks.html.

[29] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[30] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[31] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International*, 11(5), 1995.

[32] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the Annual Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, TX, January 1999.

[33] K. Nagaraja, F. Oliveria, R. Bianchini, R. P. Martin, and T. Nguyen. Understanding and dealing with operator mistakes in Internet services. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 61–76, San Francisco, CA, December 2004.

[34] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *In Proceedings of the 12th Network and Distributed Systems Security Symposium*, February 2005.

[35] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.

[36] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 177–191, October 2009.

[37] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: Improving configuration management with operating system causality analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 237–250, Stevenson, WA, October 2007.

[38] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 245–257, San Francisco, CA, December 2004.

[39] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. STRIDER: A black-box, state-based approach to change and configuration management and support. In *Proceedings of the USENIX Large Installation Systems Administration Conference*, pages 159–172, October 2003.

[40] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 77–90, San Francisco, CA, December 2004.

[41] J. Yang, C. Sar, and D. Engler. eXplode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 131–146, Seattle, WA, November 2006.

[42] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 291–304, October 2009.

[43] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–154, Pittsburgh, PA, March 2010.

[44] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 2010 European Conference on Computer Systems (EuroSys)*, pages 321–334, April 2010.

[45] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 1–14, Banff, Canada, October 2001.

[46] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieres. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

[47] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 169–180, June 2006.

[48] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards locating execution omission errors. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 415–424, June 2007.

Large-scale Incremental Processing Using Distributed Transactions and Notifications

Daniel Peng and Frank Dabek
dpeng@google.com, fdabek@google.com
Google, Inc.

Abstract

Updating an index of the web as documents are crawled requires continuously transforming a large repository of existing documents as new documents arrive. This task is one example of a class of data processing tasks that transform a large repository of data via small, independent mutations. These tasks lie in a gap between the capabilities of existing infrastructure. Databases do not meet the storage or throughput requirements of these tasks: Google's indexing system stores tens of petabytes of data and processes billions of updates per day on thousands of machines. MapReduce and other batch-processing systems cannot process small updates individually as they rely on creating large batches for efficiency.

We have built Percolator, a system for incrementally processing updates to a large data set, and deployed it to create the Google web search index. By replacing a batch-based indexing system with an indexing system based on incremental processing using Percolator, we process the same number of documents per day, while reducing the average age of documents in Google search results by 50%.

1 Introduction

Consider the task of building an index of the web that can be used to answer search queries. The indexing system starts by crawling every page on the web and processing them while maintaining a set of invariants on the index. For example, if the same content is crawled under multiple URLs, only the URL with the highest PageRank [28] appears in the index. Each link is also inverted so that the anchor text from each outgoing link is attached to the page the link points to. Link inversion must work across duplicates: links to a duplicate of a page should be forwarded to the highest PageRank duplicate if necessary.

This is a bulk-processing task that can be expressed as a series of MapReduce [13] operations: one for clustering duplicates, one for link inversion, etc. It's easy to maintain invariants since MapReduce limits the parallelism of the computation; all documents finish one processing step before starting the next. For example, when the indexing system is writing inverted links to the current highest-PageRank URL, we need not worry about its PageRank concurrently changing; a previous MapReduce step has already determined its PageRank.

Now, consider how to update that index after recrawling some small portion of the web. It's not sufficient to run the MapReduces over just the new pages since, for example, there are links between the new pages and the rest of the web. The MapReduces must be run again over the entire repository, that is, over both the new pages and the old pages. Given enough computing resources, MapReduce's scalability makes this approach feasible, and, in fact, Google's web search index was produced in this way prior to the work described here. However, reprocessing the entire web discards the work done in earlier runs and makes latency proportional to the size of the repository, rather than the size of an update.

The indexing system could store the repository in a DBMS and update individual documents while using transactions to maintain invariants. However, existing DBMSs can't handle the sheer volume of data: Google's indexing system stores tens of petabytes across thousands of machines [30]. Distributed storage systems like Bigtable [9] can scale to the size of our repository but don't provide tools to help programmers maintain data invariants in the face of concurrent updates.

An ideal data processing system for the task of maintaining the web search index would be optimized for *incremental processing*; that is, it would allow us to maintain a very large repository of documents and update it efficiently as each new document was crawled. Given that the system will be processing many small updates concurrently, an ideal system would also provide mechanisms for maintaining invariants despite concurrent updates and for keeping track of which updates have been processed.

The remainder of this paper describes a particular incremental processing system: Percolator. Percolator provides the user with random access to a multi-PB repository. Random access allows us to process documents in-

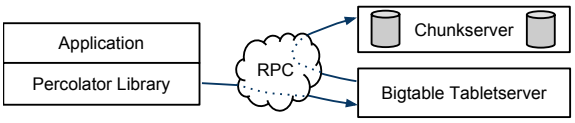


Figure 1: Percolator and its dependencies

dividually, avoiding the global scans of the repository that MapReduce requires. To achieve high throughput, many threads on many machines need to transform the repository concurrently, so Percolator provides ACID-compliant transactions to make it easier for programmers to reason about the state of the repository; we currently implement snapshot isolation semantics [5].

In addition to reasoning about concurrency, programmers of an incremental system need to keep track of the state of the incremental computation. To assist them in this task, Percolator provides observers: pieces of code that are invoked by the system whenever a user-specified column changes. Percolator applications are structured as a series of observers; each observer completes a task and creates more work for "downstream" observers by writing to the table. An external process triggers the first observer in the chain by writing initial data into the table.

Percolator was built specifically for incremental processing and is not intended to supplant existing solutions for most data processing tasks. Computations where the result can't be broken down into small updates (sorting a file, for example) are better handled by MapReduce. Also, the computation should have strong consistency requirements; otherwise, Bigtable is sufficient. Finally, the computation should be very large in some dimension (total data size, CPU required for transformation, etc.); smaller computations not suited to MapReduce or Bigtable can be handled by traditional DBMSs.

Within Google, the primary application of Percolator is preparing web pages for inclusion in the live web search index. By converting the indexing system to an incremental system, we are able to process individual documents as they are crawled. This reduced the average document processing latency by a factor of 100, and the average age of a document appearing in a search result dropped by nearly 50 percent (the age of a search result includes delays other than indexing such as the time between a document being changed and being crawled). The system has also been used to render pages into images; Percolator tracks the relationship between web pages and the resources they depend on, so pages can be reprocessed when any depended-upon resources change.

2 Design

Percolator provides two main abstractions for performing incremental processing at large scale: ACID transactions over a random-access repository and observers, a way to organize an incremental computation.

A Percolator system consists of three binaries that run on every machine in the cluster: a Percolator worker, a Bigtable [9] tablet server, and a GFS [20] chunkserver. All observers are linked into the Percolator worker, which scans the Bigtable for changed columns ("notifications") and invokes the corresponding observers as a function call in the worker process. The observers perform transactions by sending read/write RPCs to Bigtable tablet servers, which in turn send read/write RPCs to GFS chunkservers. The system also depends on two small services: the timestamp oracle and the lightweight lock service. The timestamp oracle provides strictly increasing timestamps: a property required for correct operation of the snapshot isolation protocol. Workers use the lightweight lock service to make the search for dirty notifications more efficient.

From the programmer's perspective, a Percolator repository consists of a small number of tables. Each table is a collection of "cells" indexed by row and column. Each cell contains a value: an uninterpreted array of bytes. (Internally, to support snapshot isolation, we represent each cell as a series of values indexed by timestamp.)

The design of Percolator was influenced by the requirement to run at massive scales and the lack of a requirement for extremely low latency. Relaxed latency requirements let us take, for example, a lazy approach to cleaning up locks left behind by transactions running on failed machines. This lazy, simple-to-implement approach potentially delays transaction commit by tens of seconds. This delay would not be acceptable in a DBMS running OLTP tasks, but it is tolerable in an incremental processing system building an index of the web. Percolator has no central location for transaction management; in particular, it lacks a global deadlock detector. This increases the latency of conflicting transactions but allows the system to scale to thousands of machines.

2.1 Bigtable overview

Percolator is built on top of the Bigtable distributed storage system. Bigtable presents a multi-dimensional sorted map to users: keys are (row, column, timestamp) tuples. Bigtable provides lookup and update operations on each row, and Bigtable row transactions enable atomic read-modify-write operations on individual rows. Bigtable handles petabytes of data and runs reliably on large numbers of (unreliable) machines.

A running Bigtable consists of a collection of tablet servers, each of which is responsible for serving several tablets (contiguous regions of the key space). A master coordinates the operation of tablet servers by, for example, directing them to load or unload tablets. A tablet is stored as a collection of read-only files in the Google

SSTable format. SSTables are stored in GFS; Bigtable relies on GFS to preserve data in the event of disk loss. Bigtable allows users to control the performance characteristics of the table by grouping a set of columns into a locality group. The columns in each locality group are stored in their own set of SSTables, which makes scanning them less expensive since the data in other columns need not be scanned.

The decision to build on Bigtable defined the overall shape of Percolator. Percolator maintains the gist of Bigtable's interface: data is organized into Bigtable rows and columns, with Percolator metadata stored alongside in special columns (see Figure 5). Percolator's API closely resembles Bigtable's API: the Percolator library largely consists of Bigtable operations wrapped in Percolator-specific computation. The challenge, then, in implementing Percolator is providing the features that Bigtable does not: multirow transactions and the observer framework.

2.2 Transactions

Percolator provides cross-row, cross-table transactions with ACID snapshot-isolation semantics. Percolator users write their transaction code in an imperative language (currently C++) and mix calls to the Percolator API with their code. Figure 2 shows a simplified version of clustering documents by a hash of their contents. In this example, if Commit() returns false, the transaction has conflicted (in this case, because two URLs with the same content hash were processed simultaneously) and should be retried after a backoff. Calls to Get() and Commit() are blocking; parallelism is achieved by running many transactions simultaneously in a thread pool.

While it is possible to incrementally process data without the benefit of strong transactions, transactions make it more tractable for the user to reason about the state of the system and to avoid the introduction of errors into a long-lived repository. For example, in a transactional web-indexing system the programmer can make assumptions like: the hash of the contents of a document is always consistent with the table that indexes duplicates. Without transactions, an ill-timed crash could result in a permanent error: an entry in the document table that corresponds to no URL in the duplicates table. Transactions also make it easy to build index tables that are always up to date and consistent. Note that both of these examples require transactions that span rows, rather than the single-row transactions that Bigtable already provides.

Percolator stores multiple versions of each data item using Bigtable's timestamp dimension. Multiple versions are required to provide snapshot isolation [5], which presents each transaction with the appearance of reading from a stable snapshot at some timestamp. Writes appear in a different, later, timestamp. Snapshot isolation pro-

```
bool UpdateDocument(Document doc) {
  Transaction t(&cluster);
  t.Set(doc.url(), "contents", "document", doc.contents());
  int hash = Hash(doc.contents());

  // dups table maps hash → canonical URL
  string canonical;
  if (!t.Get(hash, "canonical-url", "dups", &canonical)) {
    // No canonical yet; write myself in
    t.Set(hash, "canonical-url", "dups", doc.url());
  } // else this document already exists, ignore new copy
  return t.Commit();
}
```

Figure 2: Example usage of the Percolator API to perform basic checksum clustering and eliminate documents with the same content.

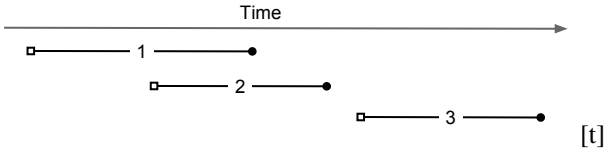


Figure 3: Transactions under snapshot isolation perform reads at a start timestamp (represented here by an open square) and writes at a commit timestamp (closed circle). In this example, transaction 2 would not see writes from transaction 1 since transaction 2's start timestamp is before transaction 1's commit timestamp. Transaction 3, however, will see writes from both 1 and 2. Transaction 1 and 2 are running concurrently: if they both write the same cell, at least one will abort.

tects against write-write conflicts: if transactions A and B, running concurrently, write to the same cell, at most one will commit. Snapshot isolation does not provide serializability; in particular, transactions running under snapshot isolation are subject to write skew [5]. The main advantage of snapshot isolation over a serializable protocol is more efficient reads. Because any timestamp represents a consistent snapshot, reading a cell requires only performing a Bigtable lookup at the given timestamp; acquiring locks is not necessary. Figure 3 illustrates the relationship between transactions under snapshot isolation.

Because it is built as a client library accessing Bigtable, rather than controlling access to storage itself, Percolator faces a different set of challenges implementing distributed transactions than traditional PDBMSs. Other parallel databases integrate locking into the system component that manages access to the disk: since each node already mediates access to data on the disk it can grant locks on requests and deny accesses that violate locking requirements.

By contrast, any node in Percolator can (and does) issue requests to directly modify state in Bigtable: there is no convenient place to intercept traffic and assign locks. As a result, Percolator must explicitly maintain locks. Locks must persist in the face of machine failure; if a lock could disappear between the two phases of com-

key	*bal:data*	*bal:lock*	*bal:write*
Bob	*6:* *5:* $10	*6:* *5:*	*6:* data @ 5 *5:*
Joe	*6:* *5:* $2	*6:* *5:*	*6:* data @ 5 *5:*

1. Initial state: Joe's account contains $2 dollars, Bob's $10.

Bob	*7:***$3** *6:* *5:* $10	*7:* **I am primary** *6:* *5:*	*7:* *6:* data @ 5 *5:*
Joe	*6:* *5:* $2	*6:* *5:*	*6:* data @ 5 *5:*

2. The transfer transaction begins by locking Bob's account balance by writing the lock column. This lock is the primary for the transaction. The transaction also writes data at its start timestamp, 7.

Bob	*7:* $3 *6:* *5:* $10	*7:* I am primary *6:* *5:*	*7:* *6:* data @ 5 *5:*
Joe	***7:* $9** *6:* *5:* $2	***7:* primary @ Bob.bal** *6:* *5:*	*7:* *6:* data @ 5 *5:*

3. The transaction now locks Joe's account and writes Joe's new balance (again, at the start timestamp). The lock is a secondary for the transaction and contains a reference to the primary lock (stored in row "Bob," column "bal"); in case this lock is stranded due to a crash, a transaction that wishes to clean up the lock needs the location of the primary to synchronize the cleanup.

Bob	*8:* *7:* $3 *6:* *5:* $10	*8:* ***7:*** *6:* *5:*	*8:* **data @ 7** *7:* *6:* data @ 5 *5:*
Joe	*7:* $9 *6:* *5:* $2	*7:* primary @ Bob.bal *6:* *5:*	*7:* *6:*data @ 5 *5:*

4. The transaction has now reached the commit point: it erases the primary lock and replaces it with a write record at a new timestamp (called the commit timestamp): 8. The write record contains a pointer to the timestamp where the data is stored. Future readers of the column "bal" in row "Bob" will now see the value $3.

Bob	*8:* *7:* $3 *6:* *5:* $10	*8:* *7:* *6:* *5:*	*8:* data @ 7 *7:* *6:* data @ 5 *5:*
Joe	*8:* *7:* $9 *6:* *5:*$2	*8:* ***7:*** *6:* *5:*	***8:* data @ 7** *7:* *6:* data @ 5 *5:*

5. The transaction completes by adding write records and deleting locks at the secondary cells. In this case, there is only one secondary: Joe.

Figure 4: This figure shows the Bigtable writes performed by a Percolator transaction that mutates two rows. The transaction transfers 7 dollars from Bob to Joe. Each Percolator column is stored as 3 Bigtable columns: data, write metadata, and lock metadata. Bigtable's timestamp dimension is shown within each cell; 12: "data" indicates that "data" has been written at Bigtable timestamp 12. Newly written data is shown in boldface.

Column	*Use*
c:lock	An uncommitted transaction is writing this cell; contains the location of primary lock
c:write	Committed data present; stores the Bigtable timestamp of the data
c:data	Stores the data itself
c:notify	Hint: observers may need to run
c:ack_O	Observer "O" has run ; stores start timestamp of successful last run

Figure 5: The columns in the Bigtable representation of a Percolator column named "c."

mit, the system could mistakenly commit two transactions that should have conflicted. The lock service must provide high throughput; thousands of machines will be requesting locks simultaneously. The lock service should also be low-latency; each Get() operation requires reading locks in addition to data, and we prefer to minimize this latency. Given these requirements, the lock server will need to be replicated (to survive failure), distributed and balanced (to handle load), and write to a persistent data store. Bigtable itself satisfies all of our requirements, and so Percolator stores its locks in special in-memory columns in the same Bigtable that stores data and reads or modifies the locks in a Bigtable row transaction when accessing data in that row.

We'll now consider the transaction protocol in more detail. Figure 6 shows the pseudocode for Percolator transactions, and Figure 4 shows the layout of Percolator data and metadata during the execution of a transaction. These various metadata columns used by the system are described in Figure 5. The transaction's constructor asks the timestamp oracle for a start timestamp (line 6), which determines the consistent snapshot seen by Get(). Calls to Set() are buffered (line 7) until commit time. The basic approach for committing buffered writes is two-phase commit, which is coordinated by the client. Transactions on different machines interact through row transactions on Bigtable tablet servers.

In the first phase of commit ("prewrite"), we try to lock all the cells being written. (To handle client failure, we designate one lock arbitrarily as the primary; we'll discuss this mechanism below.) The transaction reads metadata to check for conflicts in each cell being written. There are two kinds of conflicting metadata: if the transaction sees another write record after its start timestamp, it aborts (line 32); this is the write-write conflict that snapshot isolation guards against. If the transaction sees another lock at any timestamp, it also aborts (line 34). It's possible that the other transaction is just being slow to release its lock after having already committed below our start timestamp, but we consider this unlikely, so we abort. If there is no conflict, we write the lock and

```
1  class Transaction {
2   struct Write { Row row; Column col; string value; };
3   vector<Write> writes_;
4   int start_ts_;
5
6   Transaction() : start_ts_(oracle.GetTimestamp()) {}
7   void Set(Write w) { writes_.push_back(w); }
8   bool Get(Row row, Column c, string* value) {
9    while (true) {
10     bigtable::Txn T = bigtable::StartRowTransaction(row);
11     // Check for locks that signal concurrent writes.
12     if (T.Read(row, c+"lock", [0, start_ts_])) {
13      // There is a pending lock; try to clean it and wait
14      BackoffAndMaybeCleanupLock(row, c);
15      continue;
16     }
17
18     // Find the latest write below our start_timestamp.
19     latest_write = T.Read(row, c+"write", [0, start_ts_]);
20     if (!latest_write.found()) return false; // no data
21     int data_ts = latest_write.start_timestamp();
22     *value = T.Read(row, c+"data", [data_ts, data_ts]);
23     return true;
24    }
25   }
26  // Prewrite tries to lock cell w, returning false in case of conflict.
27   bool Prewrite(Write w, Write primary) {
28    Column c = w.col;
29    bigtable::Txn T = bigtable::StartRowTransaction(w.row);
30
31    // Abort on writes after our start timestamp ...
32    if (T.Read(w.row, c+"write", [start_ts_, ∞])) return false;
33    // ... or locks at any timestamp.
34    if (T.Read(w.row, c+"lock", [0, ∞])) return false;
35
36    T.Write(w.row, c+"data", start_ts_, w.value);
37    T.Write(w.row, c+"lock", start_ts_,
38      {primary.row, primary.col});      // The primary's location.
39    return T.Commit();
40   }
41   bool Commit() {
42    Write primary = writes_[0];
43    vector<Write> secondaries(writes_.begin()+1, writes_.end());
44    if (!Prewrite(primary, primary)) return false;
45    for (Write w : secondaries)
46     if (!Prewrite(w, primary)) return false;
47
48    int commit_ts = oracle_.GetTimestamp();
49
50    // Commit primary first.
51    Write p = primary;
52    bigtable::Txn T = bigtable::StartRowTransaction(p.row);
53    if (!T.Read(p.row, p.col+"lock", [start_ts_, start_ts_]))
54     return false;     // aborted while working
55    T.Write(p.row, p.col+"write", commit_ts,
56         start_ts_); // Pointer to data written at start_ts_
57    T.Erase(p.row, p.col+"lock", commit_ts);
58    if (!T.Commit()) return false;       // commit point
59
60    // Second phase: write out write records for secondary cells.
61    for (Write w : secondaries) {
62     bigtable::Write(w.row, w.col+"write", commit_ts, start_ts_);
63     bigtable::Erase(w.row, w.col+"lock", commit_ts);
64    }
65    return true;
66   }
67  }  // class Transaction
```

Figure 6: Pseudocode for Percolator transaction protocol.

the data to each cell at the start timestamp (lines 36-38).

If no cells conflict, the transaction may commit and proceeds to the second phase. At the beginning of the second phase, the client obtains the commit timestamp from the timestamp oracle (line 48). Then, at each cell (starting with the primary), the client releases its lock and make its write visible to readers by replacing the lock with a write record. The write record indicates to readers that committed data exists in this cell; it contains a pointer to the start timestamp where readers can find the actual data. Once the primary's write is visible (line 58), the transaction must commit since it has made a write visible to readers.

A Get() operation first checks for a lock in the timestamp range [0, start_timestamp], which is the range of timestamps visible in the transaction's snapshot (line 12). If a lock is present, another transaction is concurrently writing this cell, so the reading transaction must wait until the lock is released. If no conflicting lock is found, Get() reads the latest write record in that timestamp range (line 19) and returns the data item corresponding to that write record (line 22).

Transaction processing is complicated by the possibility of client failure (tablet server failure does not affect the system since Bigtable guarantees that written locks persist across tablet server failures). If a client fails while a transaction is being committed, locks will be left behind. Percolator must clean up those locks or they will cause future transactions to hang indefinitely. Percolator takes a lazy approach to cleanup: when a transaction A encounters a conflicting lock left behind by transaction B, A may determine that B has failed and erase its locks.

It is very difficult for A to be perfectly confident in its judgment that B is failed; as a result we must avoid a race between A cleaning up B's transaction and a not-actually-failed B committing the same transaction. Percolator handles this by designating one cell in every transaction as a synchronizing point for any commit or cleanup operations. This cell's lock is called the primary lock. Both A and B agree on which lock is primary (the location of the primary is written into the locks at all other cells). Performing either a cleanup or commit operation requires modifying the primary lock; since this modification is performed under a Bigtable row transaction, only one of the cleanup or commit operations will succeed. Specifically: before B commits, it must check that it still holds the primary lock and replace it with a write record. Before A erases B's lock, A must check the primary to ensure that B has not committed; if the primary lock is still present, then it can safely erase the lock.

When a client crashes during the second phase of commit, a transaction will be past the commit point (it has written at least one write record) but will still

have locks outstanding. We must perform roll-forward on these transactions. A transaction that encounters a lock can distinguish between the two cases by inspecting the primary lock: if the primary lock has been replaced by a write record, the transaction which wrote the lock must have committed and the lock must be rolled forward, otherwise it should be rolled back (since we always commit the primary first, we can be sure that it is safe to roll back if the primary is not committed). To roll forward, the transaction performing the cleanup replaces the stranded lock with a write record as the original transaction would have done.

Since cleanup is synchronized on the primary lock, it is safe to clean up locks held by live clients; however, this incurs a performance penalty since rollback forces the transaction to abort. So, a transaction will not clean up a lock unless it suspects that a lock belongs to a dead or stuck worker. Percolator uses simple mechanisms to determine the liveness of another transaction. Running workers write a token into the Chubby lockservice [8] to indicate they belong to the system; other workers can use the existence of this token as a sign that the worker is alive (the token is automatically deleted when the process exits). To handle a worker that is live, but not working, we additionally write the wall time into the lock; a lock that contains a too-old wall time will be cleaned up even if the worker's liveness token is valid. To handle long-running commit operations, workers periodically update this wall time while committing.

2.3 Timestamps

The timestamp oracle is a server that hands out timestamps in strictly increasing order. Since every transaction requires contacting the timestamp oracle twice, this service must scale well. The oracle periodically allocates a range of timestamps by writing the highest allocated timestamp to stable storage; given an allocated range of timestamps, the oracle can satisfy future requests strictly from memory. If the oracle restarts, the timestamps will jump forward to the maximum allocated timestamp (but will never go backwards). To save RPC overhead (at the cost of increasing transaction latency) each Percolator worker batches timestamp requests across transactions by maintaining only one pending RPC to the oracle. As the oracle becomes more loaded, the batching naturally increases to compensate. Batching increases the scalability of the oracle but does not affect the timestamp guarantees. Our oracle serves around 2 million timestamps per second from a single machine.

The transaction protocol uses strictly increasing timestamps to guarantee that Get() returns all committed writes before the transaction's start timestamp. To see how it provides this guarantee, consider a transaction R reading at timestamp T_R and a transaction W that committed at timestamp $T_W < T_R$; we will show that R sees W's writes. Since $T_W < T_R$, we know that the timestamp oracle gave out T_W before or in the same batch as T_R; hence, W requested T_W before R received T_R. We know that R can't do reads before receiving its start timestamp T_R and that W wrote locks before requesting its commit timestamp T_W. Therefore, the above property guarantees that W must have at least written all its locks before R did any reads; R's Get() will see either the fully-committed write record or the lock, in which case W will block until the lock is released. Either way, W's write is visible to R's Get().

2.4 Notifications

Transactions let the user mutate the table while maintaining invariants, but users also need a way to trigger and run the transactions. In Percolator, the user writes code ("observers") to be triggered by changes to the table, and we link all the observers into a binary running alongside every tablet server in the system. Each observer registers a function and a set of columns with Percolator, and Percolator invokes the function after data is written to one of those columns in any row.

Percolator applications are structured as a series of observers; each observer completes a task and creates more work for "downstream" observers by writing to the table. In our indexing system, a MapReduce loads crawled documents into Percolator by running loader transactions, which trigger the document processor transaction to index the document (parse, extract links, etc.). The document processor transaction triggers further transactions like clustering. The clustering transaction, in turn, triggers transactions to export changed document clusters to the serving system.

Notifications are similar to database triggers or events in active databases [29], but unlike database triggers, they cannot be used to maintain database invariants. In particular, the triggered observer runs in a separate transaction from the triggering write, so the triggering write and the triggered observer's writes are not atomic. Notifications are intended to help structure an incremental computation rather than to help maintain data integrity.

This difference in semantics and intent makes observer behavior much easier to understand than the complex semantics of overlapping triggers. Percolator applications consist of very few observers — the Google indexing system has roughly 10 observers. Each observer is explicitly constructed in the `main()` of the worker binary, so it is clear what observers are active. It is possible for several observers to observe the same column, but we avoid this feature so it is clear what observer will run when a particular column is written. Users do need to be wary about infinite cycles of notifications, but Percolator does nothing to prevent this; the user typically constructs

a series of observers to avoid infinite cycles.

We do provide one guarantee: at most one observer's transaction will commit for each change of an observed column. The converse is not true, however: multiple writes to an observed column may cause the corresponding observer to be invoked only once. We call this feature message collapsing, since it helps avoid computation by amortizing the cost of responding to many notifications. For example, it is sufficient for `http://google.com` to be reprocessed periodically rather than every time we discover a new link pointing to it.

To provide these semantics for notifications, each observed column has an accompanying "acknowledgment" column for each observer, containing the latest start timestamp at which the observer ran. When the observed column is written, Percolator starts a transaction to process the notification. The transaction reads the observed column and its corresponding acknowledgment column. If the observed column was written after its last acknowledgment, then we run the observer and set the acknowledgment column to our start timestamp. Otherwise, the observer has already been run, so we do not run it again. Note that if Percolator accidentally starts two transactions concurrently for a particular notification, they will both see the dirty notification and run the observer, but one will abort because they will conflict on the acknowledgment column. We promise that at most one observer will *commit* for each notification.

To implement notifications, Percolator needs to efficiently find dirty cells with observers that need to be run. This search is complicated by the fact that notifications are rare: our table has trillions of cells, but, if the system is keeping up with applied load, there will only be millions of notifications. Additionally, observer code is run on a large number of client processes distributed across a collection of machines, meaning that this search for dirty cells must be distributed.

To identify dirty cells, Percolator maintains a special "notify" Bigtable column, containing an entry for each dirty cell. When a transaction writes an observed cell, it also sets the corresponding notify cell. The workers perform a distributed scan over the notify column to find dirty cells. After the observer is triggered and the transaction commits, we remove the notify cell. Since the notify column is just a Bigtable column, not a Percolator column, it has no transactional properties and serves only as a hint to the scanner to check the acknowledgment column to determine if the observer should be run.

To make this scan efficient, Percolator stores the notify column in a separate Bigtable locality group so that scanning over the column requires reading only the millions of dirty cells rather than the trillions of total data cells. Each Percolator worker dedicates several threads to the scan. For each thread, the worker chooses a portion of the table to scan by first picking a random Bigtable tablet, then picking a random key in the tablet, and finally scanning the table from that position. Since each worker is scanning a random region of the table, we worry about two workers running observers on the same row concurrently. While this behavior will not cause correctness problems due to the transactional nature of notifications, it is inefficient. To avoid this, each worker acquires a lock from a lightweight lock service before scanning the row. This lock server need not persist state since it is advisory and thus is very scalable.

The random-scanning approach requires one additional tweak: when it was first deployed we noticed that scanning threads would tend to clump together in a few regions of the table, effectively reducing the parallelism of the scan. This phenomenon is commonly seen in public transportation systems where it is known as "platooning" or "bus clumping" and occurs when a bus is slowed down (perhaps by traffic or slow loading). Since the number of passengers at each stop grows with time, loading delays become even worse, further slowing the bus. Simultaneously, any bus behind the slow bus speeds up as it needs to load fewer passengers at each stop. The result is a clump of buses arriving simultaneously at a stop [19]. Our scanning threads behaved analogously: a thread that was running observers slowed down while threads "behind" it quickly skipped past the now-clean rows to clump with the lead thread and failed to pass the lead thread because the clump of threads overloaded tablet servers. To solve this problem, we modified our system in a way that public transportation systems cannot: when a scanning thread discovers that it is scanning the same row as another thread, it chooses a new random location in the table to scan. To further the transportation analogy, the buses (scanner threads) in our city avoid clumping by teleporting themselves to a random stop (location in the table) if they get too close to the bus in front of them.

Finally, experience with notifications led us to introduce a lighter-weight but semantically weaker notification mechanism. We found that when many duplicates of the same page were processed concurrently, each transaction would conflict trying to trigger reprocessing of the same duplicate cluster. This led us to devise a way to notify a cell without the possibility of transactional conflict. We implement this weak notification by writing *only* to the Bigtable "notify" column. To preserve the transactional semantics of the rest of Percolator, we restrict these weak notifications to a special type of column that cannot be written, only notified. The weaker semantics also mean that multiple observers may run and commit as a result of a single weak notification (though the system tries to minimize this occurrence). This has become an important feature for managing conflicts; if an observer

frequently conflicts on a hotspot, it often helps to break it into two observers connected by a non-transactional notification on the hotspot.

2.5 Discussion

One of the inefficiencies of Percolator relative to a MapReduce-based system is the number of RPCs sent per work-unit. While MapReduce does a single large read to GFS and obtains all of the data for 10s or 100s of web pages, Percolator performs around 50 individual Bigtable operations to process a single document.

One source of additional RPCs occurs during commit. When writing a lock, we must do a read-modify-write operation requiring two Bigtable RPCs: one to read for conflicting locks or writes and another to write the new lock. To reduce this overhead, we modified the Bigtable API by adding conditional mutations which implements the read-modify-write step in a single RPC. Many conditional mutations destined for the same tablet server can also be batched together into a single RPC to further reduce the total number of RPCs we send. We create batches by delaying lock operations for several seconds to collect them into batches. Because locks are acquired in parallel, this adds only a few seconds to the latency of each transaction; we compensate for the additional latency with greater parallelism. Batching also increases the time window in which conflicts may occur, but in our low-contention environment this has not proved to be a problem.

We also perform the same batching when reading from the table: every read operation is delayed to give it a chance to form a batch with other reads to the same tablet server. This delays each read, potentially greatly increasing transaction latency. A final optimization mitigates this effect, however: prefetching. Prefetching takes advantage of the fact that reading two or more values in the same row is essentially the same cost as reading one value. In either case, Bigtable must read the entire SSTable block from the file system and decompress it. Percolator attempts to predict, each time a column is read, what other columns in a row will be read later in the transaction. This prediction is made based on past behavior. Prefetching, combined with a cache of items that have already been read, reduces the number of Bigtable reads the system would otherwise do by a factor of 10.

Early in the implementation of Percolator, we decided to make all API calls blocking and rely on running thousands of threads per machine to provide enough parallelism to maintain good CPU utilization. We chose this thread-per-request model mainly to make application code easier to write, compared to the event-driven model. Forcing users to bundle up their state each of the (many) times they fetched a data item from the table would have made application development much more difficult. Our experience with thread-per-request was, on the whole, positive: application code is simple, we achieve good utilization on many-core machines, and crash debugging is simplified by meaningful and complete stack traces. We encountered fewer race conditions in application code than we feared. The biggest drawbacks of the approach were scalability issues in the Linux kernel and Google infrastructure related to high thread counts. Our in-house kernel development team was able to deploy fixes to address the kernel issues.

3 Evaluation

Percolator lies somewhere in the performance space between MapReduce and DBMSs. For example, because Percolator is a distributed system, it uses far more resources to process a fixed amount of data than a traditional DBMS would; this is the cost of its scalability. Compared to MapReduce, Percolator can process data with far lower latency, but again, at the cost of additional resources required to support random lookups. These are engineering tradeoffs which are difficult to quantify: how much of an efficiency loss is too much to pay for the ability to add capacity endlessly simply by purchasing more machines? Or: how does one trade off the reduction in development time provided by a layered system against the corresponding decrease in efficiency?

In this section we attempt to answer some of these questions by first comparing Percolator to batch processing systems via our experiences with converting a MapReduce-based indexing pipeline to use Percolator. We'll also evaluate Percolator with microbenchmarks and a synthetic workload based on the well-known TPC-E benchmark [1]; this test will give us a chance to evaluate the scalability and efficiency of Percolator relative to Bigtable and DBMSs.

All of the experiments in this section are run on a subset of the servers in a Google data center. The servers run the Linux operating system on x86 processors; each machine is connected to several commodity SATA drives.

3.1 Converting from MapReduce

We built Percolator to create Google's large "base" index, a task previously performed by MapReduce. In our previous system, each day we crawled several billion documents and fed them along with a repository of existing documents through a series of 100 MapReduces. The result was an index which answered user queries. Though not all 100 MapReduces were on the critical path for every document, the organization of the system as a series of MapReduces meant that each document spent 2-3 days being indexed before it could be returned as a search result.

The Percolator-based indexing system (known as Caffeine [25]), crawls the same number of documents,

but we feed each document through Percolator as it is crawled. The immediate advantage, and main design goal, of Caffeine is a reduction in latency: the median document moves through Caffeine over 100x faster than the previous system. This latency improvement grows as the system becomes more complex: adding a new clustering phase to the Percolator-based system requires an extra lookup for each document rather an extra scan over the repository. Additional clustering phases can also be implemented in the same transaction rather than in another MapReduce; this simplification is one reason the number of observers in Caffeine (10) is far smaller than the number of MapReduces in the previous system (100). This organization also allows for the possibility of performing additional processing on only a subset of the repository without rescanning the entire repository.

Adding additional clustering phases isn't free in an incremental system: more resources are required to make sure the system keeps up with the input, but this is still an improvement over batch processing systems where no amount of resources can overcome delays introduced by stragglers in an additional pass over the repository. Caffeine is essentially immune to stragglers that were a serious problem in our batch-based indexing system because the bulk of the processing does not get held up by a few very slow operations. The radically-lower latency of the new system also enables us to remove the rigid distinctions between large, slow-to-update indexes and smaller, more rapidly updated indexes. Because Percolator frees us from needing to process the repository each time we index documents, we can also make it larger: Caffeine's document collection is currently 3x larger than the previous system's and is limited only by available disk space.

Compared to the system it replaced, Caffeine uses roughly twice as many resources to process the same crawl rate. However, Caffeine makes good use of the extra resources. If we were to run the old indexing system with twice as many resources, we could either increase the index size or reduce latency by at most a factor of two (but not do both). On the other hand, if Caffeine were run with half the resources, it would not be able to process as many documents per day as the old system (but the documents it did produce would have much lower latency).

The new system is also easier to operate. Caffeine has far fewer moving parts: we run tablet servers, Percolator workers, and chunkservers. In the old system, each of a hundred different MapReduces needed to be individually configured and could independently fail. Also, the "peaky" nature of the MapReduce workload made it hard to fully utilize the resources of a datacenter compared to Percolator's much smoother resource usage.

The simplicity of writing straight-line code and the ability to do random lookups into the repository makes developing new features for Percolator easy. Under

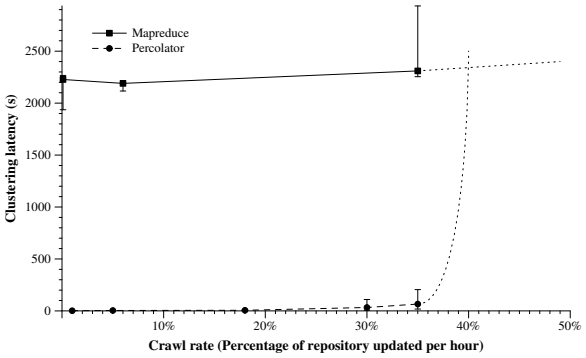


Figure 7: Median document clustering delay for Percolator (dashed line) and MapReduce (solid line). For MapReduce, all documents finish processing at the same time and error bars represent the min, median, and max of three runs of the clustering MapReduce. For Percolator, we are able to measure the delay of individual documents, so the error bars represent the 5th- and 95th-percentile delay on a per-document level.

MapReduce, random lookups are awkward and costly. On the other hand, Caffeine developers need to reason about concurrency where it did not exist in the MapReduce paradigm. Transactions help deal with this concurrency, but can't fully eliminate the added complexity.

To quantify the benefits of moving from MapReduce to Percolator, we created a synthetic benchmark that clusters newly crawled documents against a billion-document repository to remove duplicates in much the same way Google's indexing pipeline operates. Documents are clustered by three clustering keys. In a real system, the clustering keys would be properties of the document like redirect target or content hash, but in this experiment we selected them uniformly at random from a collection of 750M possible keys. The average cluster in our synthetic repository contains 3.3 documents, and 93% of the documents are in a non-singleton cluster. This distribution of keys exercises the clustering logic, but does not expose it to the few extremely large clusters we have seen in practice. These clusters only affect the latency tail and not the results we present here. In the Percolator clustering implementation, each crawled document is immediately written to the repository to be clustered by an observer. The observer maintains an index table for each clustering key and compares the document against each index to determine if it is a duplicate (an elaboration of Figure 2). MapReduce implements clustering of continually arriving documents by repeatedly running a sequence of three clustering MapReduces (one for each clustering key). The sequence of three MapReduces processes the entire repository and any crawled documents that accumulated while the previous three were running.

This experiment simulates clustering documents crawled at a uniform rate. Whether MapReduce or Percolator performs better under this metric is a function of the how frequently documents are crawled (the crawl rate)

and the repository size. We explore this space by fixing the size of the repository and varying the rate at which new documents arrive, expressed as a percentage of the repository crawled per hour. In a practical system, a very small percentage of the repository would be crawled per hour: there are over 1 trillion web pages on the web (and ideally in an indexing system's repository), far too many to crawl a reasonable fraction of in a single day. When the new input is a small fraction of the repository (low crawl rate), we expect Percolator to outperform MapReduce since MapReduce must map over the (large) repository to cluster the (small) batch of new documents while Percolator does work proportional only to the small batch of newly arrived documents (a lookup in up to three index tables per document). At very large crawl rates where the number of newly crawled documents approaches the size of the repository, MapReduce will perform better than Percolator. This cross-over occurs because streaming data from disk is much cheaper, per byte, than performing random lookups. At the cross-over the total cost of the lookups required to cluster the new documents under Percolator equals the cost to stream the documents and the repository through MapReduce. At crawl rates higher than that, one is better off using MapReduce.

We ran this benchmark on 240 machines and measured the median delay between when a document is crawled and when it is clustered. Figure 7 plots the median latency of document processing for both implementations as a function of crawl rate. When the crawl rate is low, Percolator clusters documents faster than MapReduce as expected; this scenario is illustrated by the leftmost pair of points which correspond to crawling 1 percent of documents per hour. MapReduce requires approximately 20 minutes to cluster the documents because it takes 20 minutes just to process the repository through the three MapReduces (the effect of the few newly crawled documents on the runtime is negligible). This results in an average delay between crawling a document and clustering of around 30 minutes: a random document waits 10 minutes after being crawled for the previous sequence of MapReduces to finish and then spends 20 minutes being processed by the three MapReduces. Percolator, on the other hand, finds a newly loaded document and processes it in two seconds on average, or about 1000x faster than MapReduce. The two seconds includes the time to find the dirty notification and run the transaction that performs the clustering. Note that this 1000x latency improvement could be made arbitrarily large by increasing the size of the repository.

As the crawl rate increases, MapReduce's processing time grows correspondingly. Ideally, it would be proportional to the combined size of the repository and the input which grows with the crawl rate. In practice, the running time of a small MapReduce like this is limited by stragglers, so the growth in processing time (and thus clustering latency) is only weakly correlated to crawl rate at low crawl rates. The 6 percent crawl rate, for example, only adds 150GB to a 1TB data set; the extra time to process 150GB is in the noise. The latency of Percolator is relatively unchanged as the crawl rate grows until it suddenly increases to effectively infinity at a crawl rate of 40% per hour. At this point, Percolator saturates the resources of the test cluster, is no longer able to keep up with the crawl rate, and begins building an unbounded queue of unprocessed documents. The dotted asymptote at 40% is an extrapolation of Percolator's performance beyond this breaking point. MapReduce is subject to the same effect: eventually crawled documents accumulate faster than MapReduce is able to cluster them, and the batch size will grow without bound in subsequent runs. In this particular configuration, however, MapReduce can sustain crawl rates in excess of 100% (the dotted line, again, extrapolates performance).

These results show that Percolator can process documents at orders of magnitude better latency than MapReduce in the regime where we expect real systems to operate (single-digit crawl rates).

	Bigtable	Percolator	Relative
Read/s	15513	14590	0.94
Write/s	31003	7232	0.23

Figure 8: The overhead of Percolator operations relative to Bigtable. Write overhead is due to additional operations Percolator needs to check for conflicts.

3.2 Microbenchmarks

In this section, we determine the cost of the transactional semantics provided by Percolator. In these experiments, we compare Percolator to a "raw" Bigtable. We are only interested in the relative performance of Bigtable and Percolator since any improvement in Bigtable performance will translate directly into an improvement in Percolator performance. Figure 8 shows the performance of Percolator and raw Bigtable running against a single tablet server. All data was in the tablet server's cache during the experiments and Percolator's batching optimizations were disabled.

As expected, Percolator introduces overhead relative to Bigtable. We first measure the number of random writes that the two systems can perform. In the case of Percolator, we execute transactions that write a single cell and then commit; this represents the worst case for Percolator overhead. When doing a write, Percolator incurs roughly a factor of four overhead on this benchmark. This is the result of the extra operations Percolator requires for commit beyond the single write that Bigtable issues: a read to check for locks, a write to add the lock, and a second write to remove the lock record. The read, in particular, is more expensive than a write and accounts

for most of the overhead. In this test, the limiting factor was the performance of the tablet server, so the additional overhead of fetching timestamps is not measured. We also tested random reads: Percolator performs a single Bigtable operation per read, but that read operation is somewhat more complex than the raw Bigtable operation (the Percolator read looks at metadata columns in addition to data columns).

3.3 Synthetic Workload

To evaluate Percolator on a more realistic workload, we implemented a synthetic benchmark based on TPC-E [1]. This isn't the ideal benchmark for Percolator since TPC-E is designed for OLTP systems, and a number of Percolator's tradeoffs impact desirable properties of OLTP systems (the latency of conflicting transactions, for example). TPC-E is a widely recognized and understood benchmark, however, and it allows us to understand the cost of our system against more traditional databases.

TPC-E simulates a brokerage firm with customers who perform trades, market search, and account inquiries. The brokerage submits trade orders to a market exchange, which executes the trade and updates broker and customer state. The benchmark measures the number of trades executed. On average, each customer performs a trade once every 500 seconds, so the benchmark scales by adding customers and associated data.

TPC-E traditionally has three components – a customer emulator, a market emulator, and a DBMS running stored SQL procedures. Since Percolator is a client library running against Bigtable, our implementation is a combined customer/market emulator that calls into the Percolator library to perform operations against Bigtable. Percolator provides a low-level Get/Set/iterator API rather than a high-level SQL interface, so we created indexes and did all the 'query planning' by hand.

Since Percolator is an incremental processing system rather than an OLTP system, we don't attempt to meet the TPC-E latency targets. Our average transaction latency is 2 to 5 seconds, but outliers can take several minutes. Outliers are caused by, for example, exponential backoff on conflicts and Bigtable tablet unavailability. Finally, we made a small modification to the TPC-E transactions. In TPC-E, each trade result increases the broker's commission and increments his trade count. Each broker services a hundred customers, so the average broker must be updated once every 5 seconds, which causes repeated write conflicts in Percolator. In Percolator, we would implement this feature by writing the increment to a side table and periodically aggregating each broker's increments; for the benchmark, we choose to simply omit this write.

Figure 9 shows how the resource usage of Percolator scales as demand increases. We will measure resource usage in CPU cores since that is the limiting resource in our experimental environment. We were able to procure a small number of machines for testing, but our test Bigtable cell shares the disk resources of a much larger production cluster. As a result, disk bandwidth is not a factor in the system's performance. In this experiment, we configured the benchmark with increasing numbers of customers and measured both the achieved performance and the number of cores used by all parts of the system including cores used for background maintenance such as Bigtable compactions. The relationship between performance and resource usage is essentially linear across several orders of magnitude, from 11 cores to 15,000 cores.

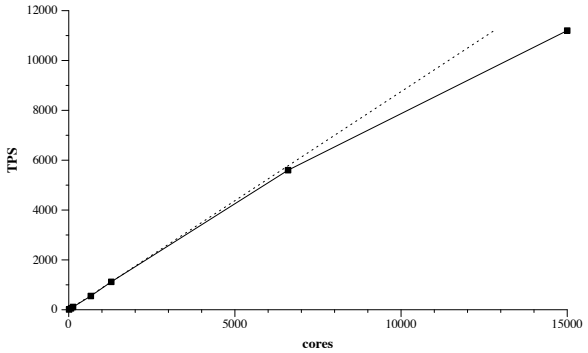


Figure 9: Transaction rate on a TPC-E-like benchmark as a function of cores used. The dotted line shows linear scaling.

This experiment also provides an opportunity to measure the overheads in Percolator relative to a DBMS. The fastest commercial TPC-E system today performs 3,183 tpsE using a single large shared-memory machine with 64 Intel Nehalem cores with 2 hyperthreads per core [33]. Our synthetic benchmark based on TPC-E performs 11,200 tps using 15,000 cores. This comparison is very rough: the Nehalem cores in the comparison machine are significantly faster than the cores in our test cell (small-scale testing on Nehalem processors shows that they are 20-30% faster per-thread compared to the cores in the test cluster). However, we estimate that Percolator uses roughly 30 times more CPU per transaction than the benchmark system. On a cost-per-transaction basis, the gap is likely much less than 30 since our test cluster uses cheaper, commodity hardware compared to the enterprise-class hardware in the reference machine.

The conventional wisdom on implementing databases is to "get close to the iron" and use hardware as directly as possible since even operating system structures like disk caches and schedulers make it hard to implement an efficient database [32]. In Percolator we not only interposed an operating system between our database and the hardware, but also several layers of software and network links. The conventional wisdom is correct: this arrangement has a cost. There are substantial overheads in

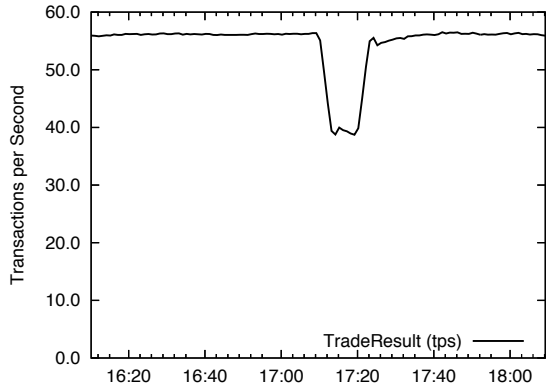


Figure 10: Recovery of tps after 33% tablet server mortality

preparing requests to go on the wire, sending them, and processing them on a remote machine. To illustrate these overheads in Percolator, consider the act of mutating the database. In a DBMS, this incurs a function call to store the data in memory and a system call to force the log to hardware controlled RAID array. In Percolator, a client performing a transaction commit sends multiple RPCs to Bigtable, which commits the mutation by logging it to 3 chunkservers, which make system calls to actually flush the data to disk. Later, that same data will be compacted into minor and major sstables, each of which will be again replicated to multiple chunkservers.

The CPU inflation factor is the cost of our layering. In exchange, we get scalability (our fastest result, though not directly comparable to TPC-E, is more than 3x the current official record [33]), and we inherit the useful features of the systems we build upon, like resilience to failures. To demonstrate the latter, we ran the benchmark with 15 tablet servers and allowed the performance to stabilize. Figure 10 shows the performance of the system over time. The dip in performance at 17:09 corresponds to a failure event: we killed a third of the tablet servers. Performance drops immediately after the failure event but recovers as the tablets are reloaded by other tablet servers. We allowed the killed tablet servers to restart so performance eventually returns to the original level.

4 Related Work

Batch processing systems like MapReduce [13, 22, 24] are well suited for efficiently transforming or analyzing an entire corpus: these systems can simultaneously use a large number of machines to process huge amounts of data quickly. Despite this scalability, re-running a MapReduce pipeline on each small batch of updates results in unacceptable latency and wasted work. Overlapping or pipelining the adjacent stages can reduce latency [10], but straggler shards still set the minimum time to complete the pipeline. Percolator avoids the expense of repeated scans by, essentially, creating indexes on the keys used to cluster documents; one of criticisms leveled by Stonebraker and DeWitt in their initial critique of MapReduce [16] was that MapReduce did not support such indexes.

Several proposed modifications to MapReduce [18, 26, 35] reduce the cost of processing changes to a repository by allowing workers to randomly read a base repository while mapping over only newly arrived work. To implement clustering in these systems, we would likely maintain a repository per clustering phase. Avoiding the need to re-map the entire repository would allow us to make batches smaller, reducing latency. DryadInc [31] attacks the same problem by reusing identical portions of the computation from previous runs and allowing the user to specify a merge function that combines new input with previous iterations' outputs. These systems represent a middle-ground between mapping over the entire repository using MapReduce and processing a single document at a time with Percolator.

Databases satisfy many of the requirements of an incremental system: a RDBMS can make many independent and concurrent changes to a large corpus and provides a flexible language for expressing computation (SQL). In fact, Percolator presents the user with a database-like interface: it supports transactions, iterators, and secondary indexes. While Percolator provides distributed transactions, it is by no means a full-fledged DBMS: it lacks a query language, for example, as well as full relational operations such as join. Percolator is also designed to operate at much larger scales than existing parallel databases and to deal better with failed machines. Unlike Percolator, database systems tend to emphasize latency over throughput since a human is often waiting for the results of a database query.

The organization of data in Percolator mirrors that of shared-nothing parallel databases [7, 15, 4]. Data is distributed across a number of commodity machines in shared-nothing fashion: the machines communicate only via explicit RPCs; no shared memory or shared disks are used. Data stored by Percolator is partitioned by Bigtable into tablets of contiguous rows which are distributed among machines; this mirrors the declustering performed by parallel databases.

The transaction management of Percolator builds on a long line of work on distributed transactions for database systems. Percolator implements snapshot isolation [5] by extending multi-version timestamp ordering [6] across a distributed system using two-phase commit.

An analogy can be drawn between the role of observers in Percolator to incrementally move the system towards a "clean" state and the incremental maintenance of materialized views in traditional databases (see Gupta and Mumick [21] for a survey of the field). In practice, while some indexing tasks like clustering documents by

contents could be expressed in a form appropriate for incremental view maintenance it would likely be hard to express the transformation of a raw document into an indexed document in such a form.

The utility of parallel databases and, by extension, a system like Percolator, has been questioned several times [17] over their history. Hardware trends have, in the past, worked against parallel databases. CPUs have become so much faster than disks that a few CPUs in a shared-memory machine can drive enough disk heads to service required loads without the complexity of distributed transactions: the top TPC-E benchmark results today are achieved on large shared-memory machines connected to a SAN. This trend is beginning to reverse itself, however, as the enormous datasets like those Percolator is intended to process become far too large for a single shared-memory machine to handle. These datasets require a distributed solution that can scale to 1000s of machines, while existing parallel databases can utilize only 100s of machines [30]. Percolator provides a system that is scalable enough for Internet-sized datasets by sacrificing some (but not all) of the flexibility and low-latency of parallel databases.

Distributed storage systems like Bigtable have the scalability and fault-tolerance properties of MapReduce but provide a more natural abstraction for storing a repository. Using a distributed storage system allows for low-latency updates since the system can change state by mutating the repository rather than rewriting it. However, Percolator is a data transformation system, not only a data storage system: it provides a way to structure computation to transform that data. In contrast, systems like Dynamo [14], Bigtable, and PNUTS [11] provide highly available data storage without the attendant mechanisms of transformation. These systems can also be grouped with the NoSQL databases (MongoDB [27], to name one of many): both offer higher performance and scale better than traditional databases, but provide weaker semantics.

Percolator extends Bigtable with multi-row, distributed transactions, and it provides the observer interface to allow applications to be structured around notifications of changed data. We considered building the new indexing system directly on Bigtable, but the complexity of reasoning about concurrent state modification without the aid of strong consistency was daunting. Percolator does not inherit all of Bigtable's features: it has limited support for replication of tables across data centers, for example. Since Bigtable's cross data center replication strategy is consistent only on a per-tablet basis, replication is likely to break invariants between writes in a distributed transaction. Unlike Dynamo and PNUTS which serve responses to users, Percolator is willing to accept the lower availability of a single data center in return for stricter consistency.

Several research systems have, like Percolator, extended distributed storage systems to include strong consistency. Sinfonia [3] provides a transactional interface to a distributed repository. Earlier published versions of Sinfonia [2] also offered a notification mechanism similar to the Percolator's observer model. Sinfonia and Percolator differ in their intended use: Sinfonia is designed to build distributed infrastructure while Percolator is intended to be used directly by applications (this probably explains why Sinfonia's authors dropped its notification mechanism). Additionally, Sinfonia's mini-transactions have limited semantics compared to the transactions provided by RDBMSs or Percolator: the user must specify a list of items to compare, read, and write prior to issuing the transaction. The mini-transactions are sufficient to create a wide variety of infrastructure but could be limiting for application builders.

CloudTPS [34], like Percolator, builds an ACID-compliant datastore on top of a distributed storage system (HBase [23] or Bigtable). Percolator and CloudTPS systems differ in design, however: the transaction management layer of CloudTPS is handled by an intermediate layer of servers called local transaction managers that cache mutations before they are persisted to the underlying distributed storage system. By contrast, Percolator uses clients, directly communicating with Bigtable, to coordinate transaction management. The focus of the systems is also different: CloudTPS is intended to be a backend for a website and, as such, has a stronger focus on latency and partition tolerance than Percolator.

ElasTraS [12], a transactional data store, is architecturally similar to Percolator; the Owning Transaction Managers in ElasTraS are essentially tablet servers. Unlike Percolator, ElasTraS offers limited transactional semantics (Sinfonia-like mini-transactions) when dynamically partitioning the dataset and has no support for structuring computation.

5 Conclusion and Future Work

We have built and deployed Percolator and it has been used to produce Google's websearch index since April, 2010. The system achieved the goals we set for reducing the latency of indexing a single document with an acceptable increase in resource usage compared to the previous indexing system.

The TPC-E results suggest a promising direction for future investigation. We chose an architecture that scales linearly over many orders of magnitude on commodity machines, but we've seen that this costs a significant 30-fold overhead compared to traditional database architectures. We are very interested in exploring this tradeoff and characterizing the nature of this overhead: how much is fundamental to distributed storage systems, and how much can be optimized away?

Acknowledgments

Percolator could not have been built without the assistance of many individuals and teams. We are especially grateful to the members of the indexing team, our primary users, and the developers of the many pieces of infrastructure who never failed to improve their services to meet our increasingly large demands.

References

[1] TPC benchmark E standard specification version 1.9.0. Tech. rep., Transaction Processing Performance Council, September 2009.

[2] AGUILERA, M. K., KARAMANOLIS, C., MERCHANT, A., SHAH, M., AND VEITCH, A. Building distributed applications using Sinfonia. Tech. rep., Hewlett-Packard Labs, 2006.

[3] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07* (2007), ACM, pp. 159–174.

[4] BARU, C., FECTEAU, G., GOYAL, A., HSIAO, H.-I., JHINGRAN, A., PADMANABHAN, S., WILSON, W., AND I HSIAO, A. G. H. DB2 parallel edition, 1995.

[5] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ANSI SQL isolation levels. In *SIGMOD* (New York, NY, USA, 1995), ACM, pp. 1–10.

[6] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Computer Surveys 13*, 2 (1981), 185–221.

[7] BORAL, H., ALEXANDER, W., CLAY, L., COPELAND, G., DANFORTH, S., FRANKLIN, M., HART, B., SMITH, M., AND VALDURIEZ, P. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering 2*, 1 (1990), 4–24.

[8] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *7th OSDI* (Nov. 2006).

[9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *7th OSDI* (Nov. 2006), pp. 205–218.

[10] CONDIE, T., CONWAY, N., ALVARO, P., AND HELLERSTIEN, J. M. MapReduce online. In *7th NSDI* (2010).

[11] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s hosted data serving platform. In *Proceedings of VLDB* (2008).

[12] DAS, S., AGRAWAL, D., AND ABBADI, A. E. ElasTraS: An elastic transactional data store in the cloud. In *USENIX HotCloud* (June 2009).

[13] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *6th OSDI* (Dec. 2004), pp. 137–150.

[14] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *SOSP '07* (2007), pp. 205–220.

[15] DEWITT, D., GHANDEHARIZADEH, S., SCHNEIDER, D., BRICKER, A., HSIAO, H.-I., AND RASMUSSEN, R. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering 2* (1990), 44–62.

[16] DEWITT, D., AND STONEBRAKER, M. MapReduce: A major step backwards. http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/.

[17] DEWITT, D. J., AND GRAY, J. Parallel database systems: the future of database processing or a passing fad? *SIGMOD Rec. 19*, 4 (1990), 104–112.

[18] EKANAYAKE, J., LI, H., ZHANG, B., GUNARATHNE, T., BAE, S.-H., QIU, J., AND FOX, G. Twister: A runtime for iterative MapReduce. In *The First International Workshop on MapReduce and its Applications* (2010).

[19] GERSHENSON, C., AND PINEDA, L. A. Why does public transport not arrive on time? The pervasiveness of equal headway instability. *PLoS ONE 4*, 10 (10 2009).

[20] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. vol. 37, pp. 29–43.

[21] GUPTA, A., AND MUMICK, I. S. Maintenance of materialized views: Problems, techniques, and applications, 1995.

[22] Hadoop. http://hadoop.apache.org/.

[23] HBase. http://hbase.apache.org/.

[24] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys '07* (New York, NY, USA, 2007), ACM, pp. 59–72.

[25] IYER, S., AND CUTTS, M. Help test some next-generation infrstructure. http://googlewebmastercentral.blogspot.com/2009/08/help-test-some-next-generation.html, August 2009.

[26] LOGOTHETIS, D., OLSTON, C., REED, B., WEBB, K. C., AND YOCUM, K. Stateful bulk processing for incremental analytics. In *SoCC '10: Proceedings of the 1st ACM symposium on cloud computing* (2010), pp. 51–62.

[27] MongoDB. http://mongodb.org/.

[28] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank citation ranking: Bringing order to the web. Tech. rep., Stanford Digital Library Technologies Project, 1998.

[29] PATON, N. W., AND DÍAZ, O. Active database systems. *ACM Computing Surveys 31*, 1 (1999), 63–103.

[30] PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., AND STONEBRAKER, M. A comparison of approaches to large-scale data analysis. In *SIGMOD '09* (June 2009), ACM.

[31] POPA, L., BUDIU, M., YU, Y., AND ISARD, M. DryadInc: Reusing work in large-scale computations. In *USENIX workshop on Hot Topics in Cloud Computing* (2009).

[32] STONEBRAKER, M. Operating system support for database management. *Communications of the ACM 24*, 7 (1981), 412–418.

[33] NEC Express5800/A1080a-E TPC-E results. http://www.tpc.org/tpce/results/tpce_result_detail.asp?id=110033001, Mar. 2010.

[34] WEI, Z., PIERRE, G., AND CHI, C.-H. CloudTPS: Scalable transactions for Web applications in the cloud. Tech. Rep. IR-CS-053, Vrije Universiteit, Amsterdam, The Netherlands, Feb. 2010. http://www.globule.org/publi/CSTWAC_ircs53.html.

[35] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *2nd USENIX workshop on Hot Topics in Cloud Computing* (2010).

Reining in the Outliers in Map-Reduce Clusters using Mantri

Ganesh Ananthanarayanan[†◇] Srikanth Kandula[†] Albert Greenberg[†]
Ion Stoica[◇] Yi Lu[†] Bikas Saha[‡] Edward Harris[‡]
[†]*Microsoft Research* [◇] *UC Berkeley* [‡] *Microsoft Bing*

Abstract– Experience from an operational Map-Reduce cluster reveals that outliers significantly prolong job completion. The causes for outliers include run-time contention for processor, memory and other resources, disk failures, varying bandwidth and congestion along network paths and, imbalance in task workload. We present Mantri, a system that monitors tasks and culls outliers using *cause-* and *resource-aware* techniques. Mantri's strategies include restarting outliers, network-aware placement of tasks and protecting outputs of valuable tasks. Using real-time progress reports, Mantri detects and acts on outliers early in their lifetime. Early action frees up resources that can be used by subsequent tasks and expedites the job overall. Acting based on the causes and the resource and opportunity cost of actions lets Mantri improve over prior work that only duplicates the laggards. Deployment in Bing's production clusters and trace-driven simulations show that Mantri improves job completion times by 32%.

1 Introduction

In a very short time, Map-Reduce has become the dominant paradigm for large data processing on compute clusters. Software frameworks based on Map-Reduce [1, 11, 13] have been deployed on tens of thousands of machines to implement a variety of applications, such as building search indices, optimizing advertisements, and mining social networks.

While highly successful, Map-Reduce clusters come with their own set of challenges. One such challenge is the often unpredictable performance of the Map-Reduce jobs. A job consists of a set of tasks which are organized in phases. Tasks in a phase depend on the results computed by the tasks in the previous phase and can run in parallel. When a task takes longer to finish than other similar tasks, tasks in the subsequent phase are delayed. At key points in the job, a few such *outlier* tasks can prevent the rest of the job from making progress. As the size of the cluster and the size of the jobs grow, the impact of outliers increases dramatically. Addressing the outlier problem is critical to speed up job completion and improve cluster efficiency.

Even a few percent of improvement in the efficiency of a cluster consisting of tens of thousands of nodes can save millions of dollars a year. In addition, finishing production jobs quickly is a competitive advantage. Doing so predictably allows SLAs to be met. In iterative modify/ debug/ analyze development cycles, the ability to iterate faster improves programmer productivity.

In this paper, we characterize the impact and causes of outliers by measuring a large Map-Reduce production cluster. This cluster is up to two orders of magnitude larger than those in previous publications [1, 13, 20] and exhibits a high level of concurrency due to many jobs simultaneously running on the cluster and many tasks on a machine. We find that variation in completion times among functionally similar tasks is large and that outliers inflate the completion time of jobs by 34% at median.

We identify three categories of root causes for outliers that are induced by the interplay between storage, network and structure of Map-Reduce jobs. First, *machine characteristics* play a key role in the performance of tasks. These include static aspects such as hardware reliability (e.g., disk failures) and dynamic aspects such as contention for processor, memory and other resources. Second, *network characteristics* impact the data transfer rates of tasks. Datacenter networks are over-subscribed leading to variance in congestion among different paths. Finally, the specifics of Map-Reduce leads to *imbalance* in work – partitioning data over a low entropy key space often leads to a skew in the input sizes of tasks.

We present Mantri[1], a system that monitors tasks and culls outliers based on their causes. It uses the following techniques: (i) Restarting outlier tasks cognizant of resource constraints and work imbalances, (ii) Network-aware placement of tasks, and (iii) Protecting output of tasks based on a cost-benefit analysis.

The detailed analysis and decision process employed by Mantri is a key departure from the state-of-the-art for outlier mitigation in Map-Reduce implementations [11, 13, 20]; these focus only on duplicating tasks. To our knowledge, none of them protect against data loss induced recomputations or network congestion induced outliers. Mantri places tasks based on the locations of their data sources as well as the current utilization of network links. On a task's completion, Mantri replicates its output if the

[1]From Sanskrit, a minister who keeps the king's court in order

benefit of not having to recompute outweighs the cost of replication.

Further, Mantri performs intelligent restarting of outliers. A task that runs for long because it has more work to do will not be restarted; if it lags due to reading data over a low-bandwidth path, it will be restarted only if a more advantageous network location becomes available. Unlike current approaches that duplicate tasks only at the end of a phase, Mantri uses real-time progress reports to act early. While early action on outliers frees up resources that could be used for pending tasks, doing so is nontrivial. A duplicate may finish faster than the original task but has the opportunity cost of consuming resources that other pending work could have used.

In summary we make the following contributions. First, we provide an analysis of the causes of outliers in a large production Map-Reduce cluster. Second, we develop Mantri, that takes early actions based on understanding the causes and the opportunity cost of actions. Finally, we perform an extensive evaluation of Mantri and compare it to existing solutions.

Mantri runs live in all of Bing's production clusters since May 2010. Results from a deployment of Mantri on a production cluster of thousands of servers and from replaying several thousand jobs collected on this cluster in a simulator show that:

- Mantri reduces the completion time of jobs by 32% on average on the production clusters. Extensive simulations show that job phases are quicker by 21% and 42% at the 50th and 75th percentiles. Mantri's median reduction in completion time improves on the next best scheme by 3.1x while using fewer resources.
- By placing *reduce* tasks to avoid network hotspots, Mantri improves the completion times of the reduce phases by 60%.
- By preferentially replicating the output of tasks that are more likely to be lost or expensive to recompute, Mantri speeds up half of the jobs by at least 20% each while only increasing the network traffic by 1%.

2 Background

We monitored the cluster and software systems that support the Bing search engine for over twelve months. This is a cluster of tens of thousands of commodity servers managed by Cosmos [8], a proprietary upgraded form of Dryad [13]. Despite a few differences, implementations of Map-Reduce [1, 8, 11, 13] are broadly similar.

Most of the jobs in the examined cluster are written in Scope [8], a mash-up language that mixes SQL-like declarative statements with user code. The Scope compiler transforms a job into a workflow– a directed acyclic graph where each node is a phase and each edge joins a phase that produces data to another that uses it. A phase is a set of one or more tasks that run in parallel and perform the same computation on different parts of the input stream. Typical phases are map, reduce and join. The number of tasks in a phase is chosen at compile time. A task will read its input over the network if it is not available on the local disk but outputs are written to the local disk. The eventual outputs of a job (as well as raw data) are stored in a reliable block storage system implemented on the same servers that do computation. Blocks are replicated n-ways for reliability. A run-time scheduler assigns tasks to machines, based on data locations, dependence patterns and cluster-wide resource availability. The network layout provides more bandwidth within a rack than across racks.

We obtain detailed logs from the Scope compiler and the Cosmos scheduler. At each of the job, phase and task levels, we record the execution behavior as represented by begin and end times, the machines(s) involved, the sizes of input and output data, the fraction of data that was read across racks and a code denoting the success or type of failure. We also record the workflow of jobs. Table 1 depicts the random subset of logs that we analyze here. Spanning eighteen days, this dataset is at least one order of magnitude larger than prior published data along many dimensions, e.g., number of jobs, cluster size.

3 The Outlier Problem

We begin with a first principles approach to the outlier problem, then analyze data from the production cluster to quantify the problem and obtain a breakdown of the causes of outliers (§4). Beginning at the first principles motivates a distinct approach (§5) which as we show in §6 significantly improves on prior art.

3.1 Outliers in a Phase

Assume a phase consists of n tasks and has s slots. Slot is a virtual token, akin to a quota, for sharing cluster resources among multiple jobs. One task can run per slot at a time. On our cluster, the median ratio of $\frac{n}{s}$ is 2.11 with a stdev of 12.37. The goal is to minimize the phase completion time, *i.e.*, the time when the last task finishes.

Based on data from the production cluster, we model t_i, the completion time of task i, as a function of the size of the data it processes, the code it runs, the resources available on the machine it executes and the bandwidth available on the network paths involved:

$$t_i = f\,(\text{datasize}, \text{code}, \text{machine}, \text{network})\,. \quad (1)$$

Large variation exists along each of the four variables leading to considerable difference in task completion times. The amount of data processed by tasks in the same phase varies, sometimes widely, due to limitations in dividing work evenly. The code is the same for tasks in a

Dates 2009-'10	Phases x 10^3	Jobs	Compute (years)	Data (PB)	Network (PB)
May 25,26	19.0	938	49.1	12.6	.66
Jun 16,17	16.5	991	88.0	22.7	1.22
Jul 20,21	22.0	1183	51.6	14.3	.67
Aug 20,21	29.2	1873	60.6	18.7	.76
Sep 15,16	27.4	1653	73.0	22.8	.73
Oct 15,16	20.4	1362	84.1	25.3	.86
Nov 16,17	37.8	1834	88.4	25.0	.68
Dec 10,11	18.7	1777	96.2	18.6	.72
Jan 11,12	24.4	1842	79.5	21.5	1.99

Table 1: Details of the logs from a production cluster consisting of thousands of servers.

phase, but differs significantly across phases (*e.g.*, map and reduce). Placing a task on a machine that has other resource hungry tasks inflates completion time, as does reading data across congested links.

In the ideal scenario, where every task takes the same amount of time, say T, scheduling is simple. Any work-conserving schedule would complete the phase in $\left(\lceil \frac{n}{s} \rceil \times T\right)$. When the task completion time varies, however, a naive work-conserving scheduler can take up to $\left(\frac{\sum_n t_i}{s} + \max t_i\right)$. A large variation in t_i increases the term $\max t_i$ and manifests as outliers.

The goal of a scheduler is to minimize the phase completion time and make it closer to $\frac{\sum_n t_i}{s}$. Sometimes, it can do even better. By placing tasks at less congested machines or network locations, the t_i's themselves can be lowered. The challenge lies in recognizing the aspects that can be changed and scheduling accordingly.

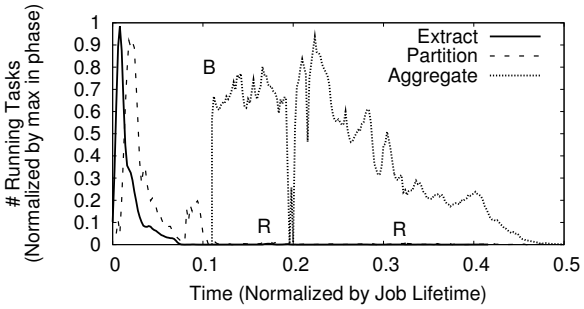
3.2 Extending from a phase to a job

The phase structure of Map-Reduce jobs adds to the variability. An outlier in an early phase, by delaying when tasks that use its output may start, has cumulative effects on the job. At *barriers* in the workflow, where none of the tasks in successive phase(s) can begin until all of the tasks in the preceding phase(s) finish, even one outlier can bring the job to a standstill[2]. Barriers occur primarily due to reduce operations that are neither commutative nor associative [28], for instance, a reduce that computes the median of records that have the same key. In our cluster, the median job workflow has eight phases and eleven edges, 47% are barriers (number of edges exceeds the number of phases due to table joins).

Dependency across phases also leads to outliers when task output is lost and needs to be *recomputed*. Data loss happens due to a combination of disk errors, software errors (*e.g.*, bugs in garbage collectors) and timeouts due to machines going unresponsive at times of high load. In fact, recomputes cause some of the longest waiting times observed on the production cluster. A recompute can cascade into earlier phases if the inputs for the recomputed task are no longer available and need to be regenerated.

[2]There is a variant in implementation where a slot is reserved for a task before all its inputs are ready. This is either to amortize the latency of network transfer by moving data over the network as soon as it is generated [1, 11], or compute partial results and present answers *online* even before the job is complete [9]. Regardless, pre-allocation of slots hogs resources for longer periods if the input task(s) straggle.

(a) Partial workflow with the number of tasks in each phase



(b) Time lapse of task execution (R=Recomputes, B=Barrier).

Figure 1: An example job from the production cluster

3.3 Illustration of Outliers

Figure 1(a) shows the workflow for a job whose structure is typical of those in the cluster. The job reads a dataset of search usage and derives an index. It consists of two Map-Reduce operations and a join, but for clarity we only show the first Map-Reduce here. Phase names follow the Dryad [13] convention– *extract* reads raw blocks, *partition* divides data on the key and *aggregate* reduces items that share a key.

Figure 1(b) depicts a timeline of an execution of this workflow. It plots the number of tasks of each phase that are active, normalized by the maximum tasks active at any time in that phase, over the lifetime of the job. Tasks in the first two phases start in quick succession to each other at ∼.05, whereas the third starts after a barrier.

Some of the outliers are evident in the long lulls before a phase ends when only a few of its tasks are active. In particular, note the regions before x∼.1 and x∼.5. The spike in phase #2 here is due to the outliers in phase #1 holding on to the job's slots. At the barrier, x∼.1, just a few outliers hold back the job from making forward progress. Though most aggregate tasks finish at x∼.3, the phase persists for another 20%.

The worst cases of waiting immediately follow recomputations of lost intermediate data marked by R. Recomputations manifest as tiny blips near the x axes for phases that had finished earlier, *e.g.*, phase #2 sees recomputes at x∼.2 though it finished at x∼.1. At x∼.2, note that aggregate almost stops due to a few recomputations.

We now quantify the magnitude of the outlier problem, before presenting our solution in detail.

4 Quantifying the Outlier Problem

We characterize the prevalence and causes of outliers and their impact on job completion times and cluster resource usage. We will argue that three factors – dynamics, concurrency and scale, that are somewhat unique to large Map-Reduce clusters for efficient and economic operation, lie at the core of the outlier problem. To our knowledge, we are the first to report detailed experiences from a large production Map-Reduce cluster.

4.1 Prevalence of Outliers

Figure 2(a) plots the fraction of high runtime outliers and recomputes in a phase. For exposition, we arbitrarily say that a task has high runtime if its time to finish is longer than 1.5x the median task duration in its phase. By recomputes, we mean instances where a task output is lost and dependent tasks wait until the output is regenerated.

We see in Figure 2(a) that 25% of phases have more than 15% of their tasks as outliers. The figure also shows that 99% of the phases see no recomputes. Though rare, recomputes have a widespread impact (§4.3). Two out of a thousand phases have over 50% of their tasks waiting for data to be recomputed.

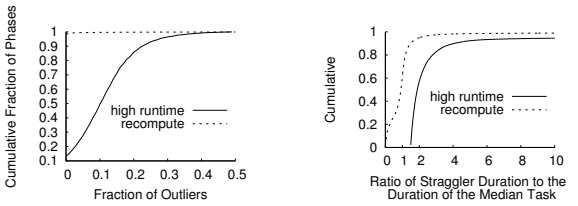
How much longer do outliers run for? Figure 2(b) shows that 80% of the runtime outliers last less than 2.5 times the phase's median task duration, with a uniform probability of being delayed by between 1.5x to 2.5x. The tail is heavy and long– 10% take more than 10x the median duration. Ignoring these if they happen early in a phase, as current approaches do, appears wasteful.

Figure 2(b) shows that most recomputations behave normally, 90% of them are clustered about the median task, but 3% take over 10x longer.

4.2 Causes of Outliers

To tease apart the contributions of each cause, we first determine whether a task's runtime can be explained by the amount of data it processes or reads across the network[3]. If yes, then the outlier is likely due to workload imbalance or poor placement. Otherwise, the outlier is likely due to resource contention or problematic machines.

Figure 3(a) shows that in 40% of the phases (top right), all the tasks with high runtimes (i.e., over 1.5x the median task) are well explained by the amount of data they process or move on the network. Duplicating these tasks would not make them run faster and will waste resources. At the other extreme, in 18% of the phases (bottom left), none of the high runtime tasks are explained by the data they process. Figure 3(b) shows tasks that take longer than they should, as predicted by the model, but do not take over 1.5x the median task in their phase. Such tasks present an opportunity for improvement. They may finish faster if run elsewhere, yet current schemes do nothing for them. 20% of the phases (on the top right) have over 55% of such improvable tasks.



(a) What fraction of tasks in a phase are outliers?

(b) How much longer do outliers take to finish?

Figure 2: Prevalence of Outliers.

Data Skew: It is natural to ask why data size varies across tasks in a phase. Across phases, the coefficient of variation ($\frac{stdev}{mean}$) in data size is .34 and 3.1 at the 50^{th} and 90^{th} percentiles. From experience, dividing work evenly is non-trivial for a few reasons. First, scheduling each additional task has overhead at the job manager. Network bandwidth is another reason. There might be too much data on a machine for a task to process, but it may be worse to split the work into multiple tasks and move data over the network. A third reason is poor coding practice. If the data is partitioned on a key space that has too little entropy, i.e., a few keys correspond to a lot of data, then the partitions will differ in size. Some reduce tasks are not amenable to splitting (neither commutative nor associative [27]), and hence each partition has to be processed by one task. Some joins and sorts are similarly constrained. Duplicating tasks that run for long because they have a lot of work to do is counter-productive.

Crossrack Traffic: Reduce phases contribute over 70% of the cross rack traffic in the cluster, while most of the rest is due to joins. We focus on cross rack traffic because the links upstream of the racks have less bandwidth than the cumulative capacity of servers in the rack.

We find that crossrack traffic leads to outliers in two ways. First, in phases where moving data across racks is avoidable (through locality constraints), a task that ends up in a disadvantageous network location runs slower than others. Second, in phases where moving data across racks is unavoidable, not accounting for the competition among tasks within the phase (self-interference) leads to outliers. In a reduce phase, for example, each task reads

[3]For each phase, we fit a linear regression model for task lifetime given the size of input and the volume of traffic moved across low bandwidth links. When the residual error for a task is less than 20%, i.e., its run time is within [.8, 1.2]x of the time predicted by this model, we call it explainable.

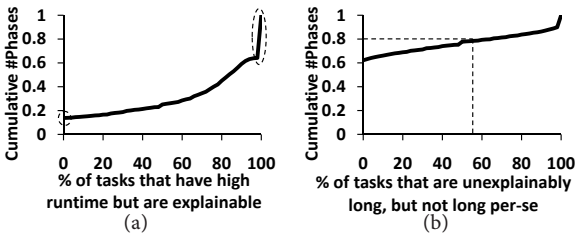


Figure 3: Contribution of data size to task runtime (see §4.2)

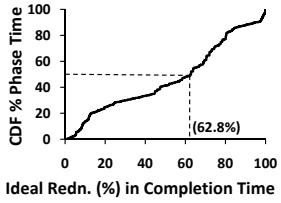


Figure 4: For reduce phases, the reduction in completion time over the current placement by placing tasks in a network-aware fashion.

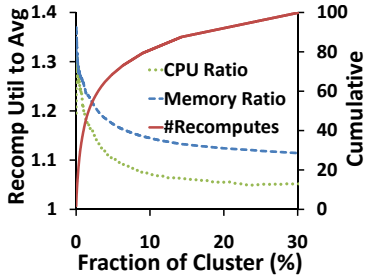


Figure 5: The ratio of processor and memory usage when recomputations happen to the average at that machine (y1). Also, the cumulative percentage of recomputations across machines (y2).

from every map task. Since the maps are spread across the cluster, regardless of where a reduce task is placed, it will read a lot of data from other racks. Current implementations place reduce tasks on any machine with spare slots. A rack that has too many reduce tasks will be congested on its downlink leading to outliers.

Figure 4 compares the current placement with an ideal one that minimizes the impact of network transfer. When possible it avoids reading data across racks and if not, places tasks such that their competition for bandwidth does not result in hotspots. In over 50% of the jobs, reduce phases account for 17% of the job's lifetime. For the reduce phases, the figure shows that the median phase takes 62% longer under the current placement.

Bad and Busy Machines: We rarely find machines that persistently inflate runtimes. Recomputations, however, are more localized. Half of them happen on 5% of the machines in the cluster. Figure 5 plots the cumulative share of recomputes across machines on the axes on the right. The figure also plots the ratio of processor and memory utilization during recomputes to the overall average on that machine. The occurrence of recomputes is correlated

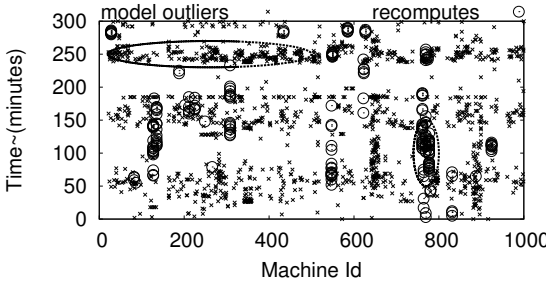


Figure 6: Clustering recomputations and outliers.

with increased use of resources by at least 20%. The subset of machines that triggers most of the recomputes is steady over days but varies over weeks, likely indicative of changing hotspots in data popularity or corruption in disks [7].

Figure 6 investigates the occurrence of "spikes" in outliers. For legibility, we only plot a subset of the machines. We find that runtime outliers (shown as stars) cluster by time. If outliers were happening at random, there should not be any horizontal bands. Rather it appears that jobs contend for resources at some times. Even at these busy times, other lightly loaded machines exist. Recomputations (shown as circles) cluster by machine. When a machine loses the output of a task, it has a higher chance of losing the output of other tasks.

Rarely does an entire rack of servers experience the same anomaly. When an anomaly happens, the fraction of other machines within the rack that see the same anomaly is less than $\frac{1}{20}$ for recomputes, and $\frac{4}{20}$ for runtime with high probability. So, it is possible to restart a task, or replicate output to protect against loss on another machine within the same rack as the original machine.

4.3 Impact of Outliers

We now examine the impact of outliers on job completion times and cluster usage. Figure 7 plots the CDF for the ratio of job completion times, with different types of outliers included, to an ideal execution that neither has skewed run times nor loses intermediate data. The y-axes weighs each job by the total cluster time its tasks take to run. The hypothetical scenarios, with some combination of outliers present but not the others, do not exist in practice. So we replayed the logs in a trace driven simulator that retains the structure of the job, the observed task durations and the probabilities of the various anomalies (details in §6). The figure shows that at median, the job completion time would be lower by 15% if runtime outliers did not happen, and by more than 34% when none of the outliers happen. Recomputations impact fewer jobs than runtime outliers, but when they do, they delay completion time by a larger amount.

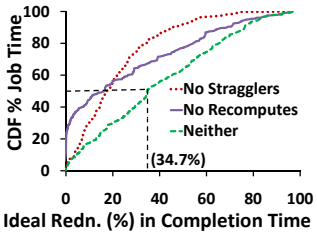


Figure 7: Percentage speed-up of job completion time in the ideal case when (some combination of) outliers do not occur.

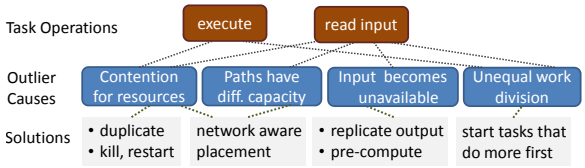


Figure 8: The Outlier Problem: Causes and Solutions

By inducing high variability in repeat runs of the same job, outliers make it hard to meet SLAs. At median, the ratio of $\frac{stdev}{mean}$ in job completion time is 0.8, i.e., jobs have a non-trivial probability of taking twice as long or finishing half as quickly.

To summarize, we take the following lessons from our experience.

- High running times of tasks do not necessarily indicate slow execution - there are multiple reasons for legitimate variation in durations of tasks.
- Every job is guaranteed some slots, as determined by cluster policy, but can use idle slots of other jobs. Hence, judicious usage of resources while mitigating outliers has collateral benefit.
- Recomputations affect jobs disproportionately. They manifest in select faulty machines and during times of heavy resource usage. Nonetheless, there are no indications of faulty racks.

5 Mantri Design

Mantri identifies points at which tasks are unable to make progress at the normal rate and implements targeted solutions. The guiding principles that distinguish Mantri from prior outlier mitigation schemes are *cause awareness* and *resource cognizance*.

Distinct actions are required for different causes. Figure 8 specifies the actions Mantri takes for each cause. If a task straggles due to contention for resources on the machine, restarting or duplicating it elsewhere can speed it up (§5.1). However, not moving data over the low bandwidth cross rack links, and if unavoidable, doing so while avoiding hotspots requires systematic placement (§5.2). To speed up tasks that wait for lost input to be recomputed, we find ways to protect task output (§5.3). Finally, for tasks with a work imbalance, we schedule the large

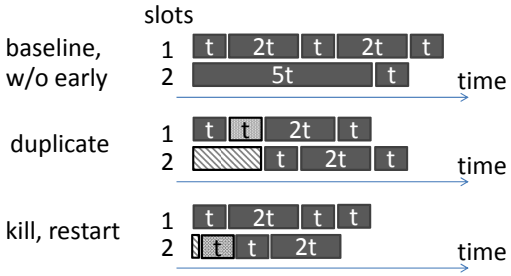


Figure 9: A stylized example to illustrate our main ideas. Tasks that are eventually killed are filled with stripes, repeat instances of a task are filled with a lighter mesh.

tasks before the others to avoid being stuck with the large ones near completion (§5.4).

There is a subtle point with outlier mitigation: reducing the completion time of a task may in fact increase the job completion time. For example, replicating the output of every task will drastically reduce recomputations–both copies are unlikely to be lost at the same time, but can slow down the job because more time and bandwidth are used up for this task denying resources to other tasks that are waiting to run. Similarly, addressing outliers early in a phase vacates slots for outstanding tasks and can speed up completion. But, potentially uses more resources per task. Unlike Mantri, none of the existing approaches act early or replicate output. Further, naively extending current schemes to act early without being cognizant of the cost of resources, as we show in §6, leads to worse performance.

Closed-loop action allows Mantri to act optimistically by bounding the cost when probabilistic predictions go awry. For example, even when Mantri cannot ascertain the cause of an outlier, it experimentally starts copies. If the cause does not repeatedly impact the task, the copy can finish faster. To handle the contrary case, Mantri continuously monitors running copies and kills those whose cost exceeds the benefit.

Based on task progress reports, Mantri estimates for each task the remaining time to finish, t_{rem}, and the predicted completion time of a new copy of the task, t_{new}. Tasks report progress once every 10s or ten times in their lifetime, whichever is smaller. We use Δ to refer to this period. We defer details of the estimation to §5.5 and proceed to describe the algorithms for mitigating each of the main causes of outliers. All that matters is that t_{rem} be an accurate estimate and that the predicted distribution t_{new} account for the underlying work that the task has to do, the appropriateness of the network location and any persistent slowness of the new machine.

5.1 Resource-aware Restart

We begin with a simple example to help exposition. Figure 9 shows a phase that has seven tasks and two slots.

```
1: let Δ = period of progress reports
2: let c = number of copies of a task
3: periodically, for each running task, kill all but the fastest α copies
       after Δ time has passed since begin
4: while slots are available do
5:     if tasks are waiting for slots then
6:         kill, restart task if t_rem > E(t_new) + Δ, stop at γ restarts
7:         duplicate if P(t_rem > t_new (c+1)/c) > δ
8:         start the waiting task that has the largest data to read
9:     else                                          ▷ all tasks have begun
10:        duplicate iff E(t_new − t_rem) > ρΔ
11:    end if
12: end while
```

Pseudocode 1: Algorithm for Resource-aware restarts (simplified).

Normal tasks run for times t and $2t$. One outlier has a runtime of $5t$. Time increases along the x axes.

The timeline at the top shows a baseline which ignores outliers and finishes at $7t$. Prior approaches that only address outliers at the end of the phase also finish at $7t$.

Note that if this outlier has a large amount of data to process letting the straggling task be is better than killing or duplicating it, both of which waste resources.

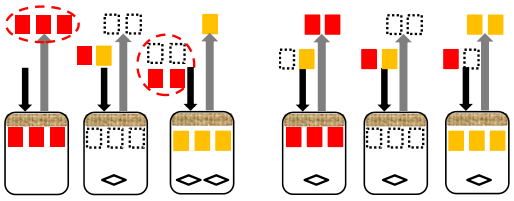
If however, the outlier was slowed down by its location, the second and third timelines compare duplication to a restart that kills the original copy. After a short time to identify the outlier, the scheduler can duplicate it at the next available slot (the middle time-line) or restart it in-place (the bottom timeline). If prediction is accurate, restarting is strictly better. However, if slots are going idle, it may be worthwhile to duplicate rather than incur the risk of losing work by killing.

Duplicating the outlier costs a total of $3t$ in resources ($2t$ before the original task is killed and t for the duplicate) which may be wasteful if the outlier were to finish in sooner than $3t$ by itself.

Restart Algorithm: Mantri uses two variants of restart, the first kills a running task and restarts it elsewhere, the second schedules a duplicate copy. In either method, Mantri restarts only when the probability of success, i.e., $\mathbb{P}(t_{new} < t_{rem})$ is high. Since t_{new} accounts for the systematic differences and the expected dynamic variation, Mantri does not restart tasks that are normal (e.g., runtime proportional to work). Pseudocode 1 summarizes the algorithm. Mantri kills and restarts a task if its remaining time is so large that there is a more than even chance that a restart would finish sooner. In particular, Mantri does so when $t_{rem} > \mathbb{E}(t_{new}) + \Delta$ [4]. To not thrash on inaccurate estimates, Mantri kills a task no more than $\gamma = 3$ times.

The "kill and restart" scheme drastically improves the job completion time without requiring extra slots as we show analytically in [5]. However, the current job scheduler incurs a queueing delay before restarting a task, that can be large and highly variant. Hence, we consider scheduling duplicates.

[4]Since the median of the heavy tailed task completion time distribution is smaller than the mean, this check implies that $\mathbb{P}(t_{new} < t_{rem}) > \mathbb{P}(t_{new} < \mathbb{E}(t_{new})) \geq .5$



(a) Ad-hoc placement (b) Even spread

Figure 10: Three reduce tasks (rhombus boxes) are to be placed across three racks. The rectangles indicate their input. The type of the rectangle indicates the map that produced this data. Each reduce task has to process one shard of each type. The ad-hoc placement on the left creates network bottlenecks on the cross-rack links (highlighted). Tasks in such racks will straggle. If the network has no other traffic, the even placement on the right avoids hotspots.

Scheduling a duplicate results in the minimum completion time of the two copies and provides a safety net when estimates are noisy or the queueing delay is large. However, it requires an extra slot and if allowed to run to finish, consumes extra computation resource that will increase the job completion time if outstanding tasks are prevented from starting. Hence, when there are outstanding tasks and no spare slots, we schedule a duplicate only if the total amount of computation resource consumed decreases. In particular, if c copies of the task are currently running, a duplicate is scheduled only if $\mathbb{P}(t_{rem} > t_{new}\frac{c+1}{c}) > \delta$. By default, $\delta = .25$. For example, a task with one running copy is duplicated only if t_{new} is less than half of t_{rem}. For stability, Mantri does not re-duplicate a task for which it launched a copy recently. Any copy that has run for some time and is slower than the second fastest copy of the task will be killed to conserve resources. Hence, there are never more than three running copies of a task[5]. When spare slots are available, as happens towards the end of the job, Mantri schedules duplicates more aggressively, i.e., whenever the reduction in the job completion time is larger than the start up time, $\mathbb{E}(t_{new} - t_{rem}) > \rho\Delta$. By default, $\rho = 3$. Note that in all the above cases, if more than one task satisfies the necessary conditions, Mantri breaks ties in favor of the task that will benefit the most.

Mantri's restart algorithm is independent of the values for its parameters. Setting γ to a larger and ρ, δ to a smaller value trades off the risk of wasteful restarts for the reward of a larger speed-up. The default values that are specified here err on the side of caution.

By scheduling duplicates conservatively and pruning aggressively, Mantri has a high success rate of its restarts. As a result, it reduces completion time and conserves resources (§6.2).

[5]The two fastest copies and the copy that has recently started.

5.2 Network-Aware Placement

Reduce tasks, as noted before (§4.2), have to read data across racks. A rack with too many reduce tasks is congested on its downlink and such tasks will straggle. Figure 10 illustrates such a scenario.

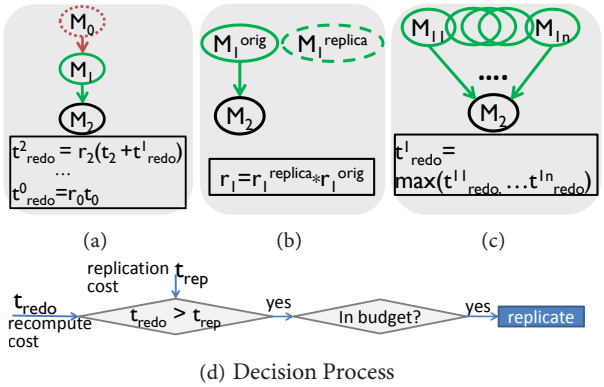
Given the utilization of all the network links and the locations of inputs for all the tasks (and jobs) that are waiting to run, optimally placing the tasks to minimize job completion time is a form of the centralized traffic engineering problem [14, 18]. However achieving up-to-date information of network state and centralized co-ordination across all jobs in the cluster are challenging. Instead, Mantri approximates the optimal placement by a local algorithm that does not track bandwidth changes nor require co-ordination across jobs.

With Mantri, each job manager places tasks so as to minimize the load on the network and avoid self-interference among its tasks. If every job manager takes this independent action, network hotspots will not cause outliers. Note that the sizes of the map outputs in each rack are known to the job manager prior to placing the tasks of the subsequent reduce phase. For a reduce phase with n tasks running on a cluster with r racks, let its input matrix $I_{n,r}$ specify the size of input in each rack for each of the tasks[6]. For any placement of reduce tasks to racks, let the data to be moved out (on the uplink) and read in (on the downlink) on the i^{th} rack be d_u^i, d_v^i, and the corresponding available bandwidths be b_u^i and b_d^i respectively. For each rack, we compute two terms $c_{2i-1} = \frac{d_u^i}{b_u^i}$ and $c_{2i} = \frac{d_v^i}{b_d^i}$. The first term is the ratio of outgoing traffic and available uplink bandwidth, and the second term is the ratio of incoming traffic and available downlink bandwidth. The algorithm computes the optimal value over all placement permutations, i.e., the rack location for each task that minimizes the maximum data transfer time, as $\arg\min \max_j c_j,\ j = 1, \cdots, 2n,$.

Rather than track the available bandwidths b_u^i and b_d^i as they change with time and as a function of other jobs in the cluster, Mantri uses these estimates. Reduce phases with a small amount of data finish quickly, and the bandwidths can be assumed to be constant throughout the execution of the phase. For phases with a large amount of data, the bandwidth averaged over their long lifetime can be assumed to be equal for all links. We see that with these estimates Mantri's placement comes close to the ideal in our experiments (see §6.4).

For phases other than reduce, Mantri complements the Cosmos policy of placing a task close to its data [23]. By accounting for the cost of moving data over low bandwidth links in t_{new}, Mantri ensures that no copy is started at a location where it has little chance of finishing earlier thereby not wasting resources.

[6] In I, the row sum indicates the data to be read by the task, whereas the column sum indicates the total input present in that rack.



(d) Decision Process

Figure 11: Avoiding costly recomputations: The cost to redo a task includes the recursive probability of predecessor tasks having to be re-done (a). Replicating output reduces the effective probability of loss (b). Tasks with many-to-one input patterns have high recomputation cost and are more valuable (c).

5.3 Avoiding Recomputation

To mitigate costly recomputations that stall a job, Mantri protects against interim data loss by replicating task output. It acts early by replicating those outputs whose cost to recompute exceeds the cost to replicate. Mantri estimates the cost to recompute as the product of the probability that the output will be lost and the time to repeat the task. The probability of loss is estimated for a machine over a long period of time. The time to repeat the task is t_{redo} with a recursive adjustment that accounts for the task's inputs also being lost. Figure 11 illustrates the calculation of t_{redo} based on the data loss probabilities (r_i's), the time taken by the tasks (t_i's) and recursively looks at prior phases. Replicating the output reduces the likelihood of recomputation to the case when all replicas are unavailable. If a task reads input from many tasks (e.g., a reduce), t_{redo} is higher since any of the inputs needing to be recomputed will stall the task's recomputation [7]. The cost to replicate, t_{rep}, is the time to move the data to another machine in the rack.

In effect, the algorithm replicates tasks at key places in a job's workflow – when the cumulative cost of not replicating many successive tasks builds up or when tasks ran on very flaky machines (high r_i) or when the output is so small that replicating it would cost little (low t_{rep}).

Further, to avoid excessive replication, Mantri limits the amount of data replicated to 10% of the data processed by the job. This limit is implemented by granting tokens proportional to the amount of data processed by each task. Task output that satisfies the above cost-benefit check is

[7] In Fig. 11(c), we assume that if multiple inputs are lost, they are recomputed in parallel and the task is stalled by the longest input. Since recomputes are rare (Fig. 2(a)), this is a fair approximation of practice.

replicated only if an equal number of tokens are available. Tokens are deducted on replication.

Mantri proactively recomputes tasks whose output and replicas, if any, have been lost. From §4, we see that recomputations on a machine cluster by time, hence Mantri considers a recompute to be the onset of a temporal problem which will cause future requests for data on this machine to fail and pre-computes such output. Doing so decreases the time that a dependent task will have to wait for lost input to be regenerated. As before, Mantri imposes a budget on the extra cluster cycles used for pre-computation. Together, probabilistic replication and pre-computation approximate the ideal scheme in our evaluation (§6.5).

5.4 Data-aware Task Ordering

Workload imbalance causes tasks to straggle. Mantri does not restart outliers that take a long time to run because they have more work to do. Instead, Mantri improves job completion time by scheduling tasks in a phase in descending order of their input size. Given n tasks, s slots and input sizes $d[1 \cdots n]$, if the optimal completion time is T_O, scheduling tasks in inverse order of their input sizes will take T, where $\frac{T}{T_O} \leq \frac{4}{3} - \frac{1}{3s}$ [12]. This means that scheduling tasks with the longest processing time first is at most 33% worse than the optimal schedule; computing the optimal is NP-hard [12].

5.5 Estimation of t_{rem} and t_{new}

Periodically, every running task informs the job scheduler of its status, including how many bytes it has read, d_{read}, thus far. Mantri combines the progress reports with the size of the input data that each task has to process, d, and predicts how much longer the task would take to finish using this model:

$$t_{rem} = t_{elapsed} \frac{d}{d_{read}} + t_{wrapup}. \quad (2)$$

The first term captures the remaining time to process data. The second term is the time to compute after all the input has been read and is estimated from the behavior of earlier tasks in the phase. Tasks may speed up or slow down and hence, rather than extrapolating from each progress report, Mantri uses a moving average. To be robust against lost progress reports, when a task hasn't reported for a while, Mantri increases t_{rem} by assuming that the task has not progressed since its last report. This linear model for estimating the remaining time for a task is well suited for data-intensive computations like Map-Reduce where a task spends most of its time reading the input data. We seldom see variance in computation time among tasks that read equal amounts of data [26].

Mantri estimates t_{new}, the distribution over time that a new copy of the task will take to run, as follows:

$$t_{new} = processRate * locationFactor * d + schedLag. \quad (3)$$

The first term is a distribution of the process rate, i.e., $\frac{\Delta time}{\Delta data}$, of all the tasks in this phase. The second term is a relative factor that accounts for whether the candidate machine for running this task is persistently slower (or faster) than other machines or has smaller (or larger) capacity on the network path to where the task's inputs are located. The third term, as before, is the amount of data the task has to process. The last term is the average delay between a task being scheduled and when it gets to run. We show in §6.2 that these estimates of t_{rem} and t_{new} are sufficiently accurate for Mantri's functioning.

6 Evaluation

We deployed and evaluated Mantri on Bing's production cluster consisting of thousands of servers. Mantri has been running as the outlier mitigation module for all the jobs in Bing's clusters since May 2010. To compare against a wider set of alternate techniques, we built a trace driven simulator that replays logs from production.

6.1 Setup

Clusters: The production cluster consists of thousands of server-class multi-core machines with tens of GBs of RAM that are spread roughly 40 servers to a rack. This cluster is used by Bing product groups. The data we analyzed earlier is from this cluster, so the observations from §4 hold here.

Workload: Mantri is the default outlier mitigation solution for the production cluster. The jobs submitted to this cluster are independent of us, enabling us to evaluate Mantri's performance in a live cluster across a variety of production jobs. We compare Mantri's performance on all jobs in the month of June 2010 with prior runs of the same jobs in April-May 2010 that ran with the earlier build of Cosmos.

In addition, we also evaluate Mantri on four hand-picked applications that represent common building blocks. *Word Count* calculates the number of unique words in the input. *Table Join* inner joins two tables each with three columns of data on one of the columns. *Group By* counts the number of occurrences of each word in the file. Finally, *grep* searches for string patterns in the input. We vary input sizes from 53 GB to 500 GB.

Prototype: Mantri builds on the Cosmos job scheduler and consists of about 1000 lines of C++ code. To compute t_{rem}, Mantri maintains an execution record for each of the running tasks that is updated when the task reports

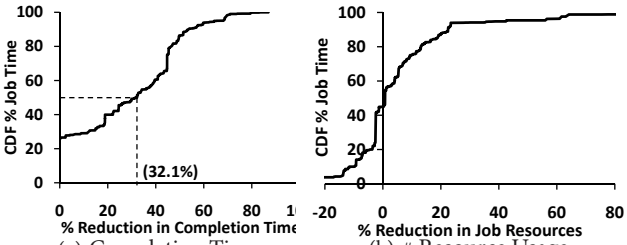
progress. A phase-wide data structure stores the necessary statistics to compute t_{new}. When slots become available, Mantri runs Pseudocode 1 and restarts or duplicates the task that would benefit the most or starts new tasks in descending order of data size. To place tasks appropriately, name builds on the per-task *affinity list*, a preferred set of machines and racks that the task can run on. At run-time the job manager attempts to place the task at its preferred locations in random order, and when none of them are available runs the task at the first available slot. The affinity list for map tasks has machines that have replicas of the input blocks. For reduce tasks, to obtain the desired proportional spread across racks (see §5.2), we populate the affinity list with a proportional number of machines in those racks.

Trace-driven Simulator: The simulator replays the logs shown in Table 1. For each phase, it faithfully repeats the observed distributions of task completion times, data read by each task, size and location of inputs, probability of failures and recomputations, and fairness based evictions. Restarted tasks have their execution times and failure probabilities sampled from the same distribution of tasks in their phase. The simulator also mimics the job workflow including semantics like barriers before phases, the permissible concurrent slots per phase and the input/output relationships between phases. It mimics cluster characteristics like machine failures, network congestion and availability of computation slots. For the network, it uses a fluid model rather than simulating individual packets. Doing the latter, at petabyte scale, is out of scope for this work.

Compared Schemes: Our results on the production cluster uses the current Dryad implementation as the baseline (§6.2). It contains state-of-the-art outlier mitigation strategies and runs thousands of jobs daily.

Our simulator performs a wider and detailed comparison. It compares Mantri with the outlier mitigation strategies in Hadoop [1], Dryad [13], Map-Reduce [11], LATE [20], and a modified form of LATE that acts on stragglers early in the phase. As the current Dryad build already has modules for straggler mitigation, we compare all of these schemes to a baseline that does not mitigate any stragglers (§6.3). On the other hand, since these schemes do not do network-aware placement or recompute mitigation, we use the current Dryad implementation itself as their baseline (§6.4 and §6.5).

We also compare Mantri against some ideal benchmarks. *NoSkew* mimics the case when all tasks in a phase take the same amount of time, set to the average over the observed task durations. *NoSkew + ChopTail* goes even further, it removes the worst quartile of the observed durations, and sets every task to the average of remaining durations. *IdealReduce* assumes perfect up-to-date knowledge of available bandwidths and places reduce tasks accordingly. *IdealRecompute* uses future knowledge of which tasks will have their inputs recomputed and protects those inputs.



(a) Completion Time (b) # Resource Usage

Figure 12: Evaluation of Mantri as the default build for all jobs on the production cluster for twenty-five days.

Metrics: As our primary metrics, we use the reduction in completion time and resource usage[8], where

$$\text{Reduction} = \frac{\text{Current} - \text{Modified}}{\text{Current}}. \quad (4)$$

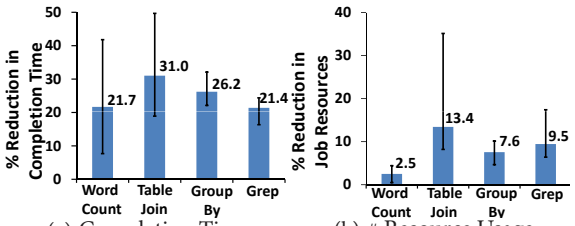
Summary: Our results are summarized as follows:

- In live deployment in the production cluster Mantri sped up the median job by 32%. 55% of the jobs experienced a net reduction in resources used. Further Mantri's network-aware placement reduced the completion times of typical reduce phases by 31%.
- Simulations driven from production logs show that Mantri's restart strategy reduces the completion time of phases by 21% (and 42%) at the 50^{th} (and 75^{th}) percentile. Here, Mantri's reduction in completion time improves on Hadoop by 3.1x while using fewer resources than Map-Reduce, each of which are the current best on those respective metrics.
- Mantri's network-aware placement of tasks speeds up half of the reduce phases by at least 60% each.
- Mantri reduces the completion times due to recomputations of jobs that constitute 25% (or 50%) of the workload by at least 40% (or 20%) each while consuming negligible extra resources.

6.2 Deployment Results

Jobs in the Wild: We compare one month of jobs in the Bing production cluster that ran after Mantri was turned live with runs of the same job(s) on earlier builds. We use only those recurring jobs that have roughly similar amounts of input and output across runs. Figure 12(a) plots the CDF of the improvement in completion time. The y axes weighs each job by the total time its tasks take to run since improvement on larger jobs adds more value

[8] A reduction of 50% implies that the property in question, completion time or resources used, decreases by half. Negative values of reduction imply that the modification uses more resources or takes longer.



(a) Completion Time (b) # Resource Usage

Figure 13: Comparing Mantri's straggler mitigation with the baseline implementation on a production cluster of thousands of servers for the four representative jobs.

	% reduction in completion time		
	avg	min	max
Phase	31.5	28.4	34.2
Job	12.6	7.0	19.2

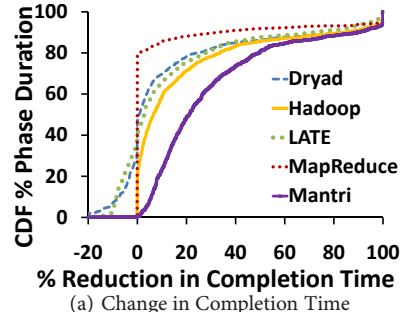
Table 2: Comparing Mantri's network-aware spread of tasks with the baseline implementation on a production cluster of thousands of servers.

to the cluster. Jobs that occupy the cluster for half the time sped up by at least 32.1%. Figure 12(b) shows that 55% of jobs see a *reduction* in resource consumption while the others use up a few extra resources. These gains are due to Mantri's ability to detect outliers early and accurately. The success rate of Mantri's copies, i.e., the fraction of time they finish before the original copy, improves by 2.8x over the earlier build. At the same time, Mantri expends fewer resources, it starts .47x fewer copies. Further, Mantri acts early, over 50% of its copies are started before the original task has completed 42% of its work as opposed to 77% with the earlier build.

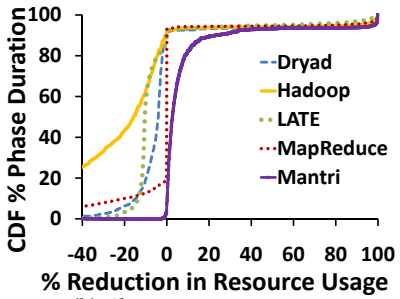
Straggler Mitigation: To cross-check the above results on standard jobs, we ran four prototypical jobs with and without Mantri twenty times each. Figure 13 shows that job completion times improve by roughly 25% and resource usage falls by roughly 10%. The histograms plot the average reduction, error bars are the 10^{th} and 90^{th} percentiles of samples. Further, we logged all the progress reports for these jobs. We find that Mantri's predictor, based on reports from the recent past, estimates t_{rem} to within a 2.9% error of the actual completion time.

Placement of Tasks: To evaluate Mantri's network-aware spreading of reduce tasks, we ran *Group By*, a job with a long-running reduce phase, ten times on the production cluster. Table 2 shows that the reduce phase's completion time reduces by 28.4% on average causing the job to speed up by an average of 12.6%. To understand why, we measure the *spread* of tasks, i.e., the ratio of the number of concurrent reduce tasks to the number of racks they ran in. High spread implies that some racks have more tasks which interfere with each other while other racks are idle. Mantri's spread is 1.5 compared to 5.5 for the earlier build.

To compare against alternative schemes and to piece apart gains from the various algorithms in Mantri, we present results from the trace-driven simulator.



(a) Change in Completion Time



(b) Change in Resource Usage

Figure 14: Comparing straggler mitigation strategies. Mantri provides a greater speed-up in completion time while using fewer resources than existing schemes.

6.3 Can Mantri mitigate stragglers?

Figure 14 compares straggler mitigation strategies in their impact on completion time and resource usage. The y-axes weighs phases by their lifetime since improving the longer phases improves cluster efficiency. The figures plot the cumulative reduction in these metrics for the 210K phases in Table 1 with each repeated thrice. For this section, our common baseline is the scheduler that takes no action on outliers. Recall from §6.1 that the simulator replays the task durations and the anomalies observed in production.

Figures 14(a) and 14(b) show that Mantri improves completion time by 21% and 42% at the 50^{th} and 75^{th} percentiles and reduces resource usage by 3% and 7% at these percentiles. From Figure 14(a), at the 50^{th} percentile, Mantri sped up phases by an additional 3.1x over the 6.9% improvement of Hadoop, the next best scheme. To achieve the smaller improvement Hadoop uses 15.9% more resources (Fig. 14(b)). Map-Reduce and Dryad have no positive impact for 80% and 50% of the phases respectively. Up to the 30^{th} percentile Dryad increases the completion time of phases. LATE is similar in its time improvement to Hadoop but uses fewer resources.

The reason for poor performance is that they miss outliers that happen early in the phase and by not knowing the true causes of outliers, the duplicates they schedule are mostly not useful. Mantri and Dryad schedule .2 restarts per task for the average phase (.06 and .56 for LATE and

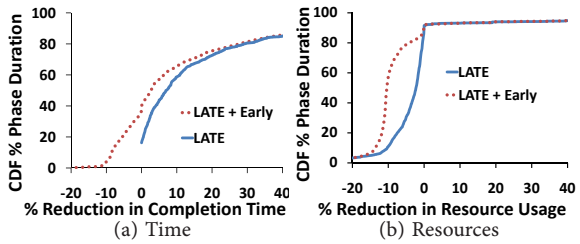


Figure 15: Extending LATE to speculate early results in worse performance

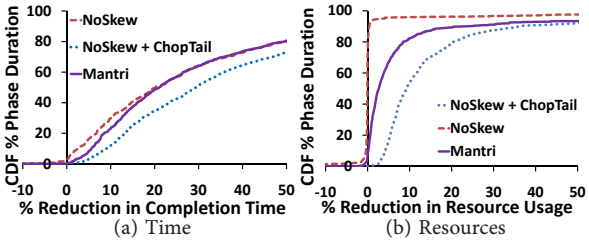


Figure 16: Mantri is on par with an ideal *NoSkew* benchmark and slightly worse than *NoSkew+ChopTail* (see end of §6.3)

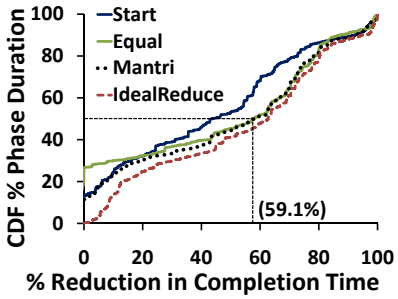


Figure 17: By being network aware, Mantri speeds up the median reduce phase by 60% over the current placement.

Hadoop). But, Mantri's restarts have a success rate of 70% compared to the 15% for LATE. The other schemes have lower success rates.

While the insight of *early action* on stragglers is valuable, it is nonetheless non trivial. We evaluate this in Figures 15(a) and 15(b) that present a form of LATE that is identical in all ways except that it addresses stragglers early. We see that addressing stragglers early increases completion time up to the 40^{th} percentile, uses more resources and is worse than vanilla LATE. Being resource aware is crucial to get the best out of early action (§5.1).

Finally, Fig. 16 shows that Mantri is on par with the ideal benchmark that has no variation in tasks, *NoSkew*, and is slightly worse than the variant that removes all durations in the top quartile, *NoSkew+ChopTail*. The reason is that Mantri's ability to substitute long running tasks with their faster copies makes up for its inability to act with perfect future knowledge of which tasks straggle.

6.4 Does Mantri improve placement?

Figure 17 plots the reduction in completion time due to Mantri's placement of reduce tasks as a CDF over all reduce phases in the dataset in Table 1. As before, the y-axes weighs phases by their lifetime. The figure shows that Mantri provides a median speed up of 59% or a 2.5x improvement over the current implementation.

The figure also compares Mantri against strategies that estimate available bandwidths differently. The *IdealReduce* strategy tracks perfectly the changes in available bandwidth of links due to the other jobs in the cluster. The *Equal* strategy assumes that the available bandwidths are equal across all links whereas *Start* assumes that the available bandwidths are the same as at the start of the phase. We see a partial order between *Start* and *Equal* (the two solid lines). Short phases are impacted by transient differences in the available bandwidths and *Start* is a good choice for these phases. However, these differences even out over the lifetime of long phases for whom *Equal* works better. Mantri is a hybrid of *Start* and *Equal*. It achieves a good approximation of *IdealReduce* without re-sampling available bandwidths.

To capture how Mantri's placement differs from Dryad, Figure 18 plots the ratio of the throughput obtained by the median task in each reduce phase to that obtained by the slowest task. With Mantri, this ratio is 1.05 at median and never larger than 2. In contrast, with Dryad's policy of placing tasks at the first available slot, this ratio is 5.25 (or 14.33) at the 50^{th} (or 75^{th}) percentile. Note that duplicating tasks that are delayed due to network congestion without considering the available bandwidths or where other tasks are located would be wasteful.

6.5 Does Mantri help with recomputations?

The best possible protection against loss of output would (a) eliminate all the increase in job completion time due to tasks waiting for their inputs to be recomputed and (b) do so with little additional cost. Mantri approximates both goals. Fig. 19 shows that Mantri achieves parity with *IdealRecompute*. Recall that IdealRecompute has perfect future knowledge of loss. The improvement in job completion time is 20% (40%) at the 50^{th} (75^{th}) percentile.

The reason is that Mantri's policy of selective replication is both accurate and biased towards the more expensive recomputations. The probability that task output that was replicated will be used because the original data becomes unavailable is 84%. Similarly, the probability that a pre-computation becomes useful is 76%, which increases to 93% if pre-computations are triggered only when two recomputations happen at a machine in quick succession. Figure 20 shows the complementary contributions from replication and pre-computation– each contribute

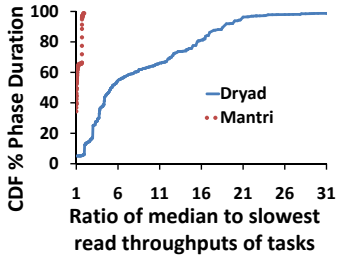


Figure 18: Unlike Dryad, Mantri's placement provides more consistent throughput to tasks in reduce phases.

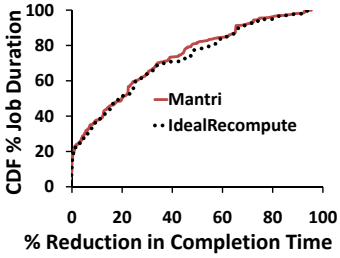


Figure 19: By probabilistically replicating task output and recomputing lost data before it is needed Mantri speeds up jobs by an amount equal to the ideal case of no data loss.

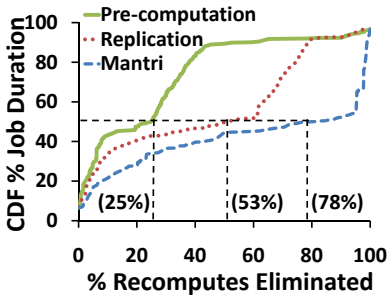


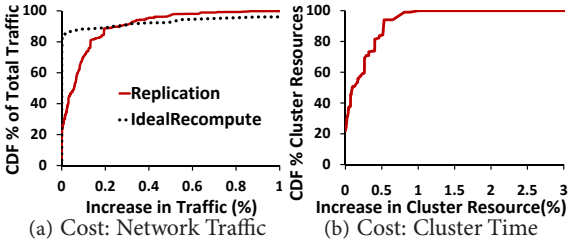
Figure 20: Fraction of recomputations that are eliminated due to Mantri's recomputation mitigation strategy, along with individual contributions from replication and pre-computation.

roughly 66% and 33% to the total. Cumulatively, the figure shows that Mantri eliminates 78% of recomputations for the median job. We note that Mantri ignores 75% of the recomputations in the bottom quartile of jobs since their impact on job completion time is small.

Fig. 21(a) shows that the extra network traffic due to replication is (overall negligible and) comparable to *IdealReduce*. Mantri sometimes replicates more data than the ideal, and at other times misses some tasks that should be replicated. Fig. 21(b) shows that pre-computations take only a few percentage extra resources.

7 Related Work

Much recent work focuses on large scale data parallel computing. Following on the Map-Reduce [11] paper, there has been work in improving workflows [1, 13], language design [8, 27], and fair schedulers [18]. Our work



(a) Cost: Network Traffic (b) Cost: Cluster Time

Figure 21: The cost to protect against recomputes is fewer than a few percentage points in both the extra traffic on the network and cluster time for pre-computation.

here takes the next step of understanding how such production clusters behave and can be improved.

Run-time stragglers have been identified by past work [11, 20]. However, we are the first to characterize the prevalence of stragglers in production and their causes. By understanding the causes, addressing stragglers early and scheduling duplicates only when there is a fair chance that the speculation saves both time and resources, our approach provides a greater reduction in job completion time while using fewer resources than prior strategies that duplicate tasks towards the end of a phase. Also, we uniquely avoid network hotspots and protect against loss of task output, two further causes of outliers.

By only acting at the end of a phase, current schemes [1, 11, 13] miss early outliers. They vary in the choice of which tasks to duplicate. After a threshold number of tasks have finished, Map-Reduce [11] duplicates all the tasks that remain. Dryad [13] duplicates those that have been running for longer than the 75th percentile of task durations. After all tasks have started, Hadoop [1] uses slots that free up to duplicate any task that has read less data than the others, while LATE [20] duplicates only those reading at a slow rate.

Though some recent proposals do away with capacity over-subscription in data centers [3, 17], today's networks remain over-subscribed albeit at smaller levels. It is common to place tasks near their input (same machine, rack etc.) for map and at the first free slot for reduce [1, 11, 13]. Our approach to eliminate outliers by a network-aware placement is orthogonal to recent work that packs tasks requiring different resources on to a machine [25], or trades-off fairness with efficiency [18]. Quincy accounts for capacity but not for runtime variations in bandwidth due to competition from other tasks.

ISS [15] protects intermediate data by replicating locally-consumed data. In particular, this does not include map output, since Hadoop transfers map output to reduce tasks as it is produced. ISS's replication strategy runs the risk of being both wasteful (when very few machines are error-prone) and insufficient (when the transfer of map output fails). In contrast, Mantri presents a broader solution that (a) replicates task output based on the probability of data loss and the recursive cost of re-

computing inputs and (b) pre-computes lost data.

The Map-Reduce paradigm is similar to parallel databases in its goal of analyzing large data [22] and to dedicated HPC clusters and parallel programs [16] by presenting similar optimization opportunities. In the context of multiple processors, studies have been done on the classic problem of dynamic task scheduling [4, 6] as well as task duplication [24]. Star-MPI [2] adapts parameters like network topology between a set of communicating processors by observing performance over time. Prior work has also focused on modeling and optimizing the communication in parallel programs [10, 19, 21] that have one-to-all or all-to-all traffic, i.e., where every receiver processes all of the output of tasks in earlier stages. In the context of the many-to-many traffic, typical of Map-Reduce, we present practical techniques for bandwidth estimation and task placement that realizes near-optimal performance.

8 Conclusion

Mantri delivers effective mitigation of outliers in Map-Reduce networks. It is motivated by, what we believe is, the first study of a large production Map-Reduce cluster. The root of Mantri's advantage lies in integrating static knowledge of job structure and dynamically available progress reports into a unified framework that identifies outliers early, applies cause-specific mitigation and does so only if the benefit is higher than the cost. In our implementation on a cluster of thousands of servers, we find Mantri to be highly effective.

Outliers are an inevitable side-effect of parallelizing work. They hurt Map-Reduce networks more due to the structure of jobs as graphs of dependent phases that pass data from one to the other. Their many causes reflect the interplay between the network, storage and, computation in Map-Reduce. Current systems shirk this complexity and assume that a duplicate would speed things up. Mantri embraces it to mitigate a broad set of outliers.

Acknowledgments– For feedback on drafts, we thank members of the RAD lab, the Cosmos product group and the OSDI reviewers. Alexei Polkhanov and Juhan Lee were invaluable in taking Mantri to production clusters.

References

[1] Hadoop distributed filesystem. http://hadoop.apache.org.

[2] A. Faraj, X. Yuan, D. Lowenthal. STAR-MPI: Self Tuned Adaptive Routines for MPI Collective Operations. In *SC*, 2006.

[3] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[4] I. Ahmad and M. K. Dhodhi. Semi-distributed load balancing for massively parallel multicomputer systems. In *IEEE TSE.*, 1991.

[5] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, and Y. Lu. Reigning in the outliers in map-reduce clusters. Technical Report MSR-TR-2010-69, Microsoft Research, 2010.

[6] B. Ucar, C. Aykanat, K. Kaya, M. Ikinci. Task assignment in Heterogeneous Computing Systems. In *JPDC*, 2006.

[7] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An analysis of data corruption in the storage stack. In *FAST*, 2008.

[8] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.

[9] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

[10] D. Culler et al. LogP: Towards a Realistic Model of Parallel Computation. In *SIGPLAN PPoPP*, 1993.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[12] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2), 1969.

[13] M. Isard et al. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Eurosys*, 2007.

[14] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. In *SIGCOMM*, 2005.

[15] S. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *SOCC*, 2010.

[16] A. Krishnamurthy and K. Yelick. Analysis and optimizations for shared address space programs. *JPDC*, 1996.

[17] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[18] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[19] M. Lauria and A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. In *JPDC*, 1997.

[20] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.

[21] P. Patarasuk, A. Faraj, X. Yuan. Pipelined Broadcast on Ethernet Switched Clusters. In *IEEE IPDPS*, 2006.

[22] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. R. Madden, and M. Stonebraker. A comparison of approaches to large scale data analysis. In *SIGMOD*, 2009.

[23] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, R. Chaiken. Nature of Datacenter Traffic: Measurements and Analysis. In *IMC*, 2009.

[24] S. Manoharan. Effect of task duplication on assignment of dependency graphs. In *Parallel Comput.*, 2001.

[25] T. Sandholm and K. Lai. Mapreduce optimization using regulated dynamic prioritization. In *SIGMETRICS*, 2009.

[26] Y. Kwon, M. Balazinska, B. Howe, J. Rolia. Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In *SOCC*, 2010.

[27] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, J. Currey. DryadLINQ: A System for General-Purpose Data-Parallel Computing Using a High-Level Language. In *OSDI*, 2008.

[28] Y. Yu, P. K. Gunda, and M. Isard. Distributed Aggregation for Data-Parallel Computing: Interfaces, Impl. In *SOSP*, 2009.

Transactional Consistency and Automatic Management in an Application Data Cache

Dan R. K. Ports Austin T. Clements Irene Zhang Samuel Madden Barbara Liskov

MIT CSAIL

txcache@csail.mit.edu

Abstract

Distributed in-memory application data caches like *memcached* are a popular solution for scaling database-driven web sites. These systems are easy to add to existing deployments, and increase performance significantly by reducing load on both the database and application servers. Unfortunately, such caches do not integrate well with the database or the application. They cannot maintain transactional consistency across the entire system, violating the isolation properties of the underlying database. They leave the application responsible for locating data in the cache and keeping it up to date, a frequent source of application complexity and programming errors.

Addressing both of these problems, we introduce a transactional cache, *TxCache*, with a simple programming model. TxCache ensures that any data seen within a transaction, whether it comes from the cache or the database, reflects a slightly stale but consistent snapshot of the database. TxCache makes it easy to add caching to an application by simply designating functions as cacheable; it automatically caches their results, and invalidates the cached data as the underlying database changes. Our experiments found that adding TxCache increased the throughput of a web application by up to $5.2\times$, only slightly less than a non-transactional cache, showing that consistency does not have to come at the price of performance.

1 Overview

Today's web applications are used by millions of users and demand implementations that scale accordingly. A typical system includes application logic (often implemented in web servers) and an underlying database that stores persistent state, either of which can become a bottleneck [1]. Increasing database capacity is typically a difficult and costly proposition, requiring careful partitioning or the use of distributed databases. Application server bottlenecks can be easier to address by adding more nodes, but this also quickly becomes expensive.

Application-level data caches, such as memcached [24], Velocity/AppFabric [34] and NCache [25], are a popular solution to server and database bottlenecks. They are deployed extensively by well-known web applications like LiveJournal, Facebook, and MediaWiki. These caches store arbitrary application-generated data in a lightweight, distributed in-memory cache. This flexibility allows an application-level cache to act as a database query cache, or to act as a web cache and cache entire web pages. But increasingly complex application logic and more personalized web content has made it more useful to cache the result of *application computations* that depend on database queries. Such caching is useful because it averts costly post-processing of database records, such as converting them to an internal representation, or generating partial HTML output. It also allows common content to be cached separately from customized content, so that it can be shared between users. For example, MediaWiki uses memcached to store items ranging from translations of interface messages to parse trees of wiki pages to the generated HTML for the site's sidebar.

Existing caches like memcached present two challenges for developers, which we address in this paper. First, they do not ensure transactional consistency with the rest of the system state. That is, there is no way to ensure that accesses to the cache and the database return values that reflect a view of the entire system at a single point in time. While the backing database goes to great length to ensure that all queries performed in a transaction reflect a consistent view of the database, *i.e.* it can ensure serializable isolation, it is nearly impossible to maintain these consistency guarantees while using a cache that operates on application objects and has no notion of database transactions. The resulting anomalies can cause incorrect information to be exposed to the user, or require more complex application logic because the application must be able to cope with violated invariants.

Second, they offer only a GET/PUT interface, placing full responsibility for explicitly managing the cache with the application. Applications must assign names to cached values, perform lookups, and keep the cache up to date. This has been a common source of programming errors in applications that use memcached. In particular, applications must explicitly *invalidate* cached data when the database changes. This is often difficult; identifying every cached application computation whose value may

have been changed requires global reasoning about the application.

We address both problems in our transactional cache, *TxCache*. TxCache provides the following features:

- transactional consistency: all data seen by the application reflects a consistent snapshot of the database, whether the data comes from cached application-level objects or directly from database queries.
- access to slightly stale but nevertheless consistent snapshots for applications that can tolerate stale data, improving cache utilization.
- a simple programming model, where applications simply designate functions as cacheable. The TxCache library then handles inserting the result of the function into the cache, retrieving that result the next time the function is called with the same arguments, and invalidating cached results when they change.

To achieve these goals, TxCache introduces the following noteworthy mechanisms:

- a protocol for ensuring that transactions see only consistent cached data, using minor database modifications to compute the validity times of database queries, and attaching them to cache objects.
- a lazy timestamp selection algorithm that assigns a transaction to a timestamp in the recent past based on the availability of cached data.
- an automatic invalidation system that tracks each object's database dependencies using dual-granularity invalidation tags, and produces notifications if they change.

We ported the RUBiS auction website prototype and MediaWiki, a popular web application, to use TxCache, and evaluated it using the RUBiS benchmark [2]. Our cache improved peak throughput by $1.5 - 5.2\times$ depending on the cache size and staleness limit, an improvement oonly slightly below that of a non-transactional cache.

The next section presents the programming model and consistency semantics. Section 3 sketches the structure of the system, and Sections 4–6 describe each component in detail. Section 7 describes our experiences porting applications to TxCache, Section 8 presents a performance evaluation, and Section 9 reviews the related work.

2 System and Programming Model

TxCache is designed for systems consisting of one or more application servers that interact with a database server. These application servers could be web servers running embedded scripts (*e.g.* with `mod_php`), or dedicated application servers, as with Sun's Enterprise Java Beans. The database server is a standard relational database; for simplicity, we assume the application uses a single database to store all of its persistent state.

TxCache introduces two new components, as shown in Figure 1: a cache and an application-side cache library, as well as some minor modifications to the database server. The cache is partitioned across a set of cache nodes, which may run on dedicated hardware or share it with other servers. The application never interacts with the cache servers; the TxCache library transparently translates an application's cacheable functions into cache accesses.

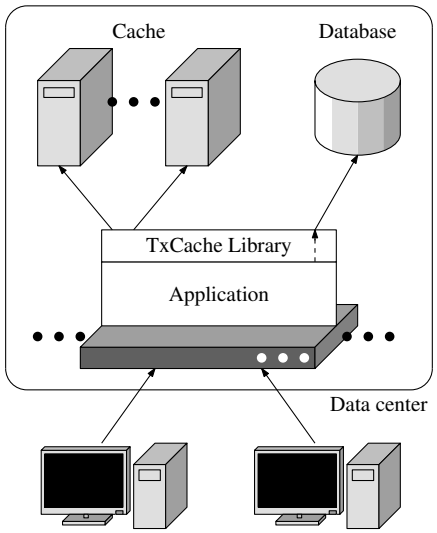


Figure 1: Key components in a TxCache deployment. The system consists of a single database, a set of cache nodes, and a set of application servers. TxCache also introduces an application library, which handles all interactions with the cache server.

2.1 Programming Model

Our goal is to make it easy to incorporate caching into a new or existing application. Towards this end, TxCache provides an application library with a simple programming model, shown in Figure 2, based on *cacheable functions*. Applications developers can cache computations simply by designating functions to be cached.

Programs group their operations into transactions. TxCache requires applications to specify whether their transactions are read-only or read/write by using either the BEGIN-RO or BEGIN-RW function. Transactions are ended by calling COMMIT or ABORT. Within a transaction block, TxCache ensures that, regardless of whether the application gets its data from the database or the cache, it sees a view consistent with the state of the database at a single point in time.

Within a transaction, operations can be grouped into *cacheable functions*. These are actual functions in the program's code, annotated to indicate that their results can be cached. A cacheable function can consist of database queries and computation, and can also make calls to other cacheable functions. To be suitable for caching, functions

- BEGIN-RO(*staleness*) : Begin a read-only transaction. The transaction sees a consistent snapshot from within the past *staleness* seconds.
- BEGIN-RW() : Begin a read/write transaction.
- COMMIT() → *timestamp* : Commit a transaction and return the timestamp at which it ran
- ABORT() : Abort a transaction

- MAKE-CACHEABLE(*fn*) → *cached-fn* : Makes a function cacheable. *cached-fn* is a new function that first checks the cache for the result of another call with the same arguments. If not found, it executes *fn* and stores its result in the cache.

Figure 2: TxCache library API

must be pure, *i.e.* they must be deterministic, not have side effects, and depend only on their arguments and the database state. For example, it would not make sense to cache a function that returns the current time. TxCache currently relies upon programmers to ensure that they only cache suitable functions, but this requirement could also be enforced using static or dynamic analysis [14, 33].

Cacheable functions are essentially memoized. TxCache's library provides a MAKE-CACHEABLE function that takes an implementation of a cacheable function and returns a wrapper function that can be called to take advantage of the cache. When called, the wrapper function checks if the cache contains the result of a previous call to the function with the same arguments that is consistent with the current transaction's snapshot. If so, it returns it. Otherwise, it invokes the implementation function and stores the returned value in the cache. With proper linguistic support (*e.g.* Python decorators), marking a function cacheable can be as simple as adding a tag to its existing definition.

Our cacheable function interface is easier to use than the GET/PUT interface provided by existing caches like memcached. It does not require programmers to manually assign keys to cached values and keep them up to date. Although seemingly straightforward, this is nevertheless a source of errors because selecting keys requires reasoning about the entire application and how the application might evolve. Examining MediaWiki bug reports, we found that several memcached-related MediaWiki bugs stemmed from choosing insufficiently descriptive keys, causing two different objects to overwrite each other [22]. In one case, a user's watchlist page was always cached under the same key, causing the same results to be returned even if the user requested to display a different number of days worth of changes.

TxCache's programming model has another crucial benefit: it does not require applications to explicitly update or invalidate cached results when modifying the database. Adding explicit invalidations requires global reasoning about the application, hindering modularity: adding caching for an object requires knowing every place it could possibly change. This, too, has been a source of bugs in MediaWiki [23]. For example, editing a wiki page clearly requires invalidating any cached copies of that page. But other, less obvious objects must be invalidated too. Once MediaWiki began storing each user's edit count in their cached USER object, it became necessary to invalidate this object after an edit. This was initially forgotten, indicating that identifying all cached objects needing invalidation is not straightforward, especially in applications so complex that no single developer is aware of the whole of the application.

2.2 Consistency Model

TxCache provides *transactional consistency*: all requests within a transaction see a consistent view of the system as of a specific timestamp. That is, requests see only the effects of other transactions that committed prior to that timestamp. For read/write transactions, TxCache supports this guarantee by running them directly on the database, bypassing the cache entirely. Read-only transactions use objects in the cache, and TxCache ensures that nevertheless they view a consistent state.

Most caches return slightly stale data simply because modified data does not reach the cache immediately. TxCache goes further by allowing applications to specify an explicit *staleness limit* to BEGIN-RO, indicating that that the transaction can see a view of data from that time or later. However, regardless of the age of the snapshot, each transaction always sees a consistent view. This feature is motivated by the observation that many applications can tolerate a certain amount of staleness [18], and using stale cached data can improve the cache's hit rate [21].

Applications can specify their staleness limit on a per-transaction basis. Additionally, when a transaction commits, TxCache provides the user with the timestamp at which it ran. Together, these can be used to avoid anomalies. For example, an application can store the timestamp of a user's last transaction in its session state, and use that as a staleness bound so that the user never observes time moving backwards. More generally, these timestamps can be used to ensure a causal ordering between related transactions [20].

We chose to have read/write transactions bypass the cache entirely so that TxCache does not introduce new anomalies. The application can expect the same guarantees (and anomalies) of the underlying database. For example, if the underlying database uses snapshot isolation, the system will still have the same anomalies as snapshot isolation, but TxCache will never introduce snapshot isolation anomalies into the read/write transactions of a system that does not use snapshot isolation. Our model

could be extended to allow read/write transactions to read information from the cache, if applications are willing to accept the risk of anomalies. One particular challenge is that read/write transactions typically expect to see the effects of their own updates, while these cannot be made visible to other transactions until the commit point.

3 System Architecture

In order to present an easy-to-use interface to application developers, TxCache needs to store cached data, keep it up to date, and ensure that data seen by an application is transactionally consistent. This section and the following ones describe how it achieves this using cache servers, modifications to the database, and an application-side library. None of this complexity, however, is visible to the application, which sees only cachable functions.

An application running with TxCache accesses information from the cache whenever possible, and from the database on a cache miss. To ensure it sees a consistent view, TxCache uses versioning. Each database query has an associated *validity interval*, describing the range of time over which its result was valid, which is computed automatically by the database. The TxCache library tracks the queries that a cached value depends on, and uses them to tag the cache entry with a validity interval. Then, the library provides consistency by ensuring that, within each read-only transaction, it only retrieves values from the cache and database that were valid at the same time. Thus, each transaction effectively sees a snapshot of the database taken at a particular time, even as it accesses data from the cache.

Section 4 describes how the cache is structured, and defines how a cached object's validity interval and database dependencies are represented. Section 5 describes how the database is modified to track query validity intervals and provide invalidation notifications when a query's result changes. Section 6 describes how the library tracks dependencies for application objects, and selects consistent values from the cache and database.

4 Cache Design

TxCache stores cached data in RAM on a number of cache servers. The cache presents a hash table interface: it maps keys to associated values. Applications do not interact with the cache directly; the TxCache library translates the name and arguments of a function call into a hash key, and checks and updates the cache itself.

Data is partitioned among cache nodes using a consistent hashing approach [17], as in peer-to-peer distributed hash tables [31, 35]. Unlike DHTs, we assume that the system is small enough that every application node can maintain a complete list of cache servers, allowing it to immediately map a key to the responsible node. This list could be maintained by hand in small systems, or using a group membership service [10] in larger or more dynamic environments.

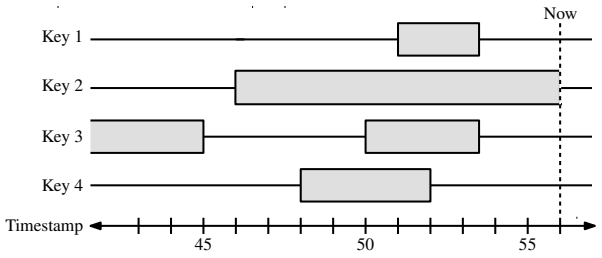


Figure 3: An example of versioned data in the cache at one point in time. Each rectangle is a version of a data item. For example, the data for key 1 became valid with commit 51 and invalid with commit 53, and the data for key 2 became valid with commit 46 and is still valid.

4.1 Versioning

Unlike a simple hash table, our cache is *versioned*. In addition to its key, each entry in the cache is tagged with its *validity interval*, as shown in Figure 3. This interval is the range of time at which the cached value was current. Its lower bound is the commit time of the transaction that caused it to become valid, and its upper bound is the commit time of the first subsequent transaction to change the result, making the cache entry invalid. The cache can store multiple cache entries with the same key; they will have disjoint validity intervals because only one is valid at any time. Whenever the TxCache library puts the result of a cacheable function call into the cache, it includes the validity interval of that result (derived using information obtained from the database).

To look up a result in the cache, the TxCache library sends both the key it is interested in and a timestamp or range of acceptable timestamps. The cache server returns a value consistent with the library's request, *i.e.* one whose validity interval intersects the given range of acceptable timestamps, if any exists. The server also returns the value's associated validity interval. If multiple such values exist, the cache server returns the most recent one.

When a cache node runs out of memory, it evicts old cached values to free up space for new ones. Cache entries are never pinned and can always be discarded; if one is later needed, it is simply a cache miss. A cache eviction policy can take into account both the time since an entry was accessed, and its staleness. Our cache server uses a least-recently-used replacement policy, but also eagerly removes any data too stale to be useful.

4.2 Invalidation Tags and Streams

When an object is inserted into the cache, it can be flagged as *still-valid* if it reflects the latest state of the database, like Key 2 in Figure 3. For such objects, the database

provides *invalidation* notifications when they change.

Every still-valid object has an associated set of *invalidation tags* that describe which parts of the database it depends on. Each invalidation tag has two parts: a table name and an optional index key description. The database identifies the invalidation tags for a query based on the access methods used to access the database. A query that uses an index equality lookup receives a two-part tag, *e.g.* a search for users with name Alice would receive tag USERS:NAME=ALICE. A query that performs a sequential scan or index range scan has a wildcard for the second part of the tag, *e.g.* USERS:⋆. Wildcard invalidations are expected to be very rare because applications typically try to perform only index lookups; they exist primarily for completeness. Queries that access multiple tables or multiple keys in a table receive multiple tags. The object's final tag set will have one or more tags for each query that the object depends on.

The database distributes invalidations to the cache as an *invalidation stream*. This is an ordered sequence of messages, one for each update transaction, containing the transaction's timestamp and all invalidation tags that it affected. Each message is delivered to all cache nodes by a reliable application-level multicast mechanism [10], or by link-level broadcast if possible. The cache servers process the messages in order, truncating the validity interval for any affected object at the transaction's timestamp.

Using the same transaction timestamps to order cache entries and invalidations eliminates race conditions that could occur if an invalidation reaches the cache server before an item is inserted with the old value. These race conditions are a real concern: MediaWiki does not cache failed article lookups, because a negative result might never be removed from the cache if the report of failure is stale but arrived after its corresponding invalidation.

For cache lookup purposes, items that are still valid are treated as though they have an upper validity bound equal to the timestamp of the last invalidation received prior to the lookup. This ensures that there is no race condition between an item being changed on the database and invalidated in the cache, and that multiple items modified by the same transaction are invalidated atomically.

5 Database Support

The validity intervals that TxCache uses in its cache are derived from validity information generated by the database. To make this possible, TxCache uses a modified DBMS that has similar versioning properties to the cache. Specifically, it can run queries on slightly stale snapshots, and it computes validity intervals for each query result it returns. It also assigns invalidation tags to queries, and produces the invalidation stream described in Section 4.2.

Though standard databases do not provide these features, we show they can be implemented by reusing the same mechanisms that are used to implement multiversion concurrency control techniques like snapshot isolation. In this section, we describe how we modified an existing DBMS, PostgreSQL [29], to provide the necessary support. The modifications are not extensive (under 2000 lines of code in our implementation). Moreover, they are not Postgres-specific; the approach can be applied to other databases that use multiversion concurrency.

5.1 Exposing Multiversion Concurrency

Because our cache allows read-only transactions to run slightly in the past, the database must be able to perform queries against a past snapshot of a database. This situation arises when a read-only transaction is assigned a timestamp in the past and reads some cached data, and then a later operation in the same transaction results in a cache miss, requiring the application to query the database. The database query must return results consistent with the cached values already seen, so the query must execute at the same timestamp in the past.

Temporal databases, which track the history of their data and allow "time travel," solve this problem but impose substantial storage and indexing cost to support complex queries over the entire history of the database. What we require is much simpler: we only need to run a transaction on a stale but recent snapshot. Our insight is that these requirements are essentially identical to those for supporting snapshot isolation [5], so many databases already have the infrastructure to support them.

We modified Postgres to expose the multiversion storage it uses internally to provide snapshot isolation. We added a PIN command that assigns an ID to a read-only transaction's snapshot. When starting a new transaction, the TxCache library can specify this ID using the new BEGIN SNAPSHOTID syntax, creating a new transaction that sees the same view of the database as the erstwhile read-only transaction. The database state for that snapshot will be retained at least until it is released by the UNPIN command. A pinned snapshot is identified by the commit time of the last committed transaction visible to it, allowing it to be easily ordered with respect to update transactions and other snapshots.

Postgres is especially well-suited to this modification because of its "no-overwrite" storage manager [36], which already retains recent versions of data. Because stale data is only removed periodically by an asynchronous "vacuum cleaner" process, the fact that we keep data around slightly longer has little impact on performance. However, our technique is not Postgres-specific; any database that implements snapshot isolation must have a way to keep a similar history of recent database states, such as Oracle's rollback segments.

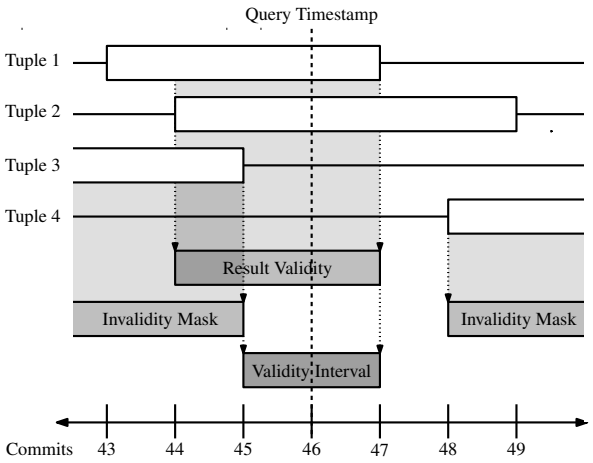


Figure 4: Example of tracking the validity interval for a read-only query. All four tuples match the query predicate. Tuples 1 and 2 match the timestamp, so their intervals intersect to form the result validity. Tuples 3 and 4 fail the visibility test, so their intervals join to form the invalidity mask. The final validity interval is the difference between the result validity and the invalidity mask.

5.2 Tracking Result Validity

TxCache needs the database server to provide the validity interval for every query result in order to ensure transactional consistency of cached objects. Recall that this is defined as the range of timestamps for which the query would give the same results. Its lower bound is the commit time of the most recent transaction that added, deleted, or modified any tuple in the result set. It may have an upper bound if a subsequent transaction changed the result, or it may be unbounded if the result is still current.

The validity interval is computed as the intersection of two ranges, the *result tuple validity* and the *invalidity mask*, which we track separately.

The result tuple validity is the intersection of the validity times of the tuples returned by the query. For example, tuple 1 in Figure 4 was deleted at time 47, and tuple 2 was created at time 44; the result would be different before time 44 or after time 47. This interval is easy to compute because multiversion concurrency requires that each tuple in the database be tagged with the ID of its creating transaction and deleting transaction (if any). We simply propagate these tags throughout query execution. If an operator, such as a join, combines multiple tuples to produce a single result, the validity interval of the output tuple is the intersection of its inputs.

The result tuple validity, however, does not completely capture the validity of a query, because of *phantoms*. These are tuples that did *not* appear in the result, but would have if the query were run at a different timestamp. For example, tuple 3 in Figure 4 will not appear in the results because it was deleted before the query timestamp, but the results would be different if the query were run before it was deleted. Similarly, tuple 4 is not visible because it was created afterwards. We capture this effect with the invalidity mask, which is the union of the validity times for all tuples that failed the *visibility check*, *i.e.* were discarded because their timestamps made them invisible to the transaction's snapshot. Throughout query execution, whenever such a tuple is encountered, its validity interval is added to the invalidity mask.

The invalidity mask is conservative because visibility checks are performed as early as possible in the query plan to avoid processing unnecessary tuples. Some of these tuples might have been discarded anyway if they failed the query conditions later in the query plan (perhaps after joining with another table). While being conservative preserves the correctness of the cached results, it might unnecessarily constrain the validity intervals of cached items, reducing the hit rate. To ameloriate this problem, we continue to perform the visibility check as early as possible, but during sequential scans and index lookups, we evaluate the predicate before the visibility check. This differs from regular Postgres with respect to sequential scans, where it evaluates the cheaper visibility check first. Delaying the visibility checks improves the quality of the invalidity mask, and incurs little overhead for simple predicates, which are most common.

Finally, the invalidity mask is subtracted from the result tuple validity to give the query's final validity interval. This interval is reported to the TxCache library, piggybacked on each SELECT query result; the library combines these intervals to obtain validity intervals for objects it stores in the cache.

5.3 Automating Invalidations

When the database executes a query and reports that its validity interval is unbounded, *i.e.* the query result is still valid, it assumes responsibility for providing an invalidation when the result may have changed. At query time, it must assign invalidation tags to indicate the query's dependencies, and at update time, it must notify the cache of invalidation tags for objects that might have changed.

When a query is performed, the database examines the query plan it generates. At the lowest level of the tree are the access methods that obtain the data, *e.g.* a sequential scan of a heap file, or a B-tree index lookup. For index equality lookups, the database assigns an invalidation tag of the form TABLE:KEY. For other types, it assigns a wildcard tag TABLE:$\star$. Each query may have multiple tags; the complete set is returned along with the SELECT query results.

When a read/write transaction modifies some tuples, the database identifies the set of invalidation tags affected.

Each tuple added, deleted, or modified yields one invalidation tag for each index it is listed in. If a transaction modifies most of a table, the database can aggregate multiple tags into a single wildcard tag on TABLE:⋆. Generated invalidation tags are queued until the transaction commits. When it does, the database server passes the set of tags, along with the transaction's timestamp, to the multicast service for distribution to the cache nodes, ensuring that the invalidation stream is properly ordered.

5.4 Pincushion

TxCache needs to keep track of which snapshots are pinned on the database, and which of those are within a read-only transaction's staleness limit. It also must eventually unpin old snapshots, provided that they are not used by running transactions. The DBMS itself could be responsible for tracking this information. However, to simplify implementation, and to reduce the overall load on the database, we placed this functionality instead in a lightweight daemon known as the *pincushion* (so named because it holds the pinned snapshot IDs). It can be run on the database host, on a cache server, or elsewhere.

The pincushion maintains a table of currently pinned snapshots, containing the snapshot's ID, the corresponding wall-clock timestamp, and the number of running transactions that might be using it. When the TxCache library running on an application node begins a read-only transaction, it requests from the pincushion all sufficiently fresh pinned snapshots, *e.g.* those pinned in the last 30 seconds. The pincushion flags these snapshots as possibly in use, for the duration of the transaction. If there are no sufficiently fresh pinned snapshots, the TxCache library starts a read-only transaction on the database, running on the latest snapshot, and pins that snapshot. It then registers the snapshot's ID and the wall-clock time (as reported by the database) with the pincushion. The pincushion also periodically scans its list of pinned snapshots, removing any unused snapshots older than a threshold by sending an UNPIN command to the database.

Though the pincushion is accessed on every transaction, it performs little computation and is unlikely to form a bottleneck. In all of our experiments, nearly all pincushion requests received a response in under 0.2 ms, approximately the network round-trip time. We have also developed a protocol for replicating the pincushion to increase its throughput, but it has yet to become necessary.

6 Cache Library

Applications interact with TxCache through its application-side library, which keeps them blissfully unaware of the details of cache servers, validity intervals, invalidation tags and the like. It is responsible for assigning timestamps to read-only transactions, retrieving values from the cache when cacheable functions are called, storing results in the cache, and computing the validity intervals and invalidation tags for anything it stores in the cache.

In this section, we describe the implementation of the TxCache library. For clarity, we begin with a simplified version where timestamps are chosen when a transaction begins and cacheable functions do not call other cacheable functions. In Section 6.2, we describe a technique for choosing timestamps lazily to take better advantage of cached data. In Section 6.3, we lift the restriction on nested calls.

6.1 Basic Functionality

The TxCache library is divided into a language-independent library that implements the core functionality, and a set of bindings that implement language-specific interfaces. Currently, we have only implemented bindings for PHP, but adding support for other languages should be relatively straightforward.

Recall from Figure 2 that the library's interface is simple: it provides the standard transaction commands (BEGIN, COMMIT, and ABORT), and functions are designated as cacheable using a MAKE-CACHEABLE function that takes a function and returns a wrapped function that first checks for available cached values[1].

When a transaction is started, the application specifies whether it is read/write or read-only, and, if read-only, the staleness limit. For a read/write transaction, the TxCache library simply starts a transaction on the database server, and passes all queries directly to it. At the beginning of a read-only transaction, the library contacts the pincushion to request the list of pinned snapshots within the staleness limit, then chooses one to run the transaction at. If no sufficiently recent snapshots exist, the library starts a new transaction on the database and pins its snapshot.

The library can delay beginning an underlying read-only transaction on the database (*i.e.* sending a BEGIN SQL statement) until it actually needs to issue a query. Thus, transactions whose requests are all satisfied from the cache do not need to connect to the database at all.

When a cacheable function's wrapper is called, the library checks whether its result is in the cache. To do so, it serializes the function's name and arguments into a key (a hash of the function's code could also be used to handle software updates). The library finds the responsible cache server using consistent hashing, and sends it a LOOKUP request. The request includes the transaction's timestamp, which any returned value must satisfy. If the cache returns a matching result, the library returns it directly to the program.

In the event of a cache miss, the library calls the cacheable function's implementation. As the cacheable

[1] In languages such as PHP that lack higher-order functions, the syntax is slightly more complicated, but the concept is the same.

function issues queries to the database, the library accumulates the validity intervals and invalidation tags returned by these queries. The final result of the cacheable function is valid at all times in the intersection of the accumulated validity intervals. When the cacheable function returns, the library serializes its result and inserts it into the cache, tagged with the accumulated validity interval and any invalidation tags.

6.2 Choosing Timestamps Lazily

Above, we assumed that the library chooses a read-only transaction's timestamp when the transaction starts. Although straightforward, this approach requires the library to decide on a timestamp without any knowledge of what data is in the cache or what data will be accessed. Lacking this knowledge, it is not clear what policy would provide the best hit rate.

However, the timestamp need not be chosen immediately. Instead, it can be chosen lazily based on which cached results are available. This takes advantage of the fact that each cached value is valid over a range of timestamps: its validity interval. For example, consider a transaction that has observed a single cached result x. This transaction can still be serialized at *any* timestamp in x's validity interval. On the transaction's next call to a cacheable function, any cached value whose validity interval overlaps x's can be chosen, as this still ensures there is at least one timestamp at which the transaction can be serialized. As the transaction proceeds, the set of possible serialization points narrows each time the transaction reads a cached value or a database query result.

Specifically, the algorithm proceeds as follows. When a transaction begins, the library requests from the pincushion all pinned snapshot IDs that satisfy its freshness requirement. It stores this set as its *pin set*. The pin set represents the set of timestamps at which the current transaction can be serialized; it will be updated as the cache and the database are accessed. The pin set also initially contains a special ID, denoted $\star$, which indicates that the transaction can also be run in the present, on some newly pinned snapshot. The pin set only contains $\star$ until the first cacheable function in the transaction executes.

When the application invokes a cacheable function, the library sends a LOOKUP request for the appropriate key, but instead of indicating a single timestamp, it indicates the *bounds* of the pin set (the lowest and highest timestamp, excluding $\star$). The transaction can use any cached value whose validity interval overlaps these bounds and still remain serializable at one or more timestamps. The library then reduces the transaction's pin set by eliminating all timestamps that do not lie in the returned value's validity interval, since observing a cached value means the transaction can no longer be serialized outside its validity interval. This includes removing $\star$ from the pin-set because once the transaction has used cached data, it cannot be run on a new, possibly inconsistent snapshot.

When the cache does not contain any entries that match both the key and the requested interval, a cache miss occurs. In this case, the library calls the cacheable function's implementation, as before. When the transaction makes its first database query, the library is finally forced to select a specific timestamp from the pin set and BEGIN a read-only transaction on the database at the chosen timestamp. If a non-$\star$ timestamp is chosen, the transaction runs on that timestamp's saved snapshot. If $\star$ is chosen, the library starts a new transaction, pinning the latest snapshot and reporting the pin to the pincushion. The pin set is then *reified*: $\star$ is replaced with the newly-created snapshot's timestamp, replacing the abstract concept of "the present time" with a concrete timestamp.

The library needs a policy to choose which pinned snapshot from the pin set it should run at. Simply choosing $\star$ if available, or the most recent timestamp otherwise, biases transactions towards running on recent data, but results in a very large number of pinned snapshots, which can ultimately slow the system down. To avoid the overhead of creating many snapshots, we used the following policy: if the most recent timestamp in the pin set is older than five seconds and $\star$ is available, then the library chooses $\star$ in order to produce a new pinned snapshot; otherwise it chooses the most recent timestamp.

During the execution of a cacheable function, the validity intervals of the queries that the function makes are accumulated, and their intersection defines the validity interval of the cacheable result, just as before. In addition, just like when a transaction observes values from the cache, each time it observes query results from the database, the transaction's pin set is reduced by eliminating all timestamps outside the result's validity interval, as the transaction can no longer be serialized at these points. If the transaction's pin set still contains $\star$, $\star$ is removed.

The validity interval of the cacheable function and pin set of the transaction are two distinct but related notions: the function's validity interval is the set of timestamps at which its result is valid, and the pin set is the set of timestamps at which the enclosing transaction can be serialized. The pin set always lies within the validity interval, but the two may differ when a transaction calls multiple cacheable functions in sequence, or performs "bare" database queries outside a cacheable function.

6.2.1 Correctness

Lazy selection of timestamps is a complex algorithm, and its correctness is not self-evident. The following two properties show that it provides transactional consistency.

Invariant 1. *All data seen by the application during a read-only transaction is consistent with the database*

state at every timestamp in the pin set, i.e. *the transaction can be serialized at any timestamp in the pin set.*

Invariant 1 holds because any timestamps *in*consistent with data the application has seen are removed from the pin set. The application sees two types of data: cached values and database query results. Each is tagged with its validity interval. The library removes from the pin set all timestamps that lie outside either of these intervals.

Invariant 2. *The pin set is never empty,* i.e. *the transaction can always be serialized at some timestamp.*

The pin set is initially non-empty: it contains the timestamps of all sufficiently-fresh pinned snapshots, if any, and always $\star$. So we must ensure that at least one timestamp remains every time the pin set shrinks, *i.e.* when a result is obtained from the cache or database.

When a value is fetched from the cache, its validity interval is guaranteed to intersect the transaction's pin set at at least one timestamp. The cache will only return an entry with a non-empty intersection between its validity interval and the bounds of the transaction's pin set. This intersection contains the timestamp of at least one pinned snapshot: if the result's validity interval lies partially within and partially outside the bounds of the client's pin set, then either the earliest or latest timestamp in the pin set lies in the intersection. If the result's validity interval lies entirely within the bounds of the transaction's pin set, then the pin set contains at least the timestamp of the pinned snapshot from which the cached result was originally generated. Thus, Invariant 2 continues to hold even after removing from the pin set any timestamps that do not lie within the cached result's validity interval.

It is easier to see that when the database returns a query result, the validity interval intersects the pin set at at least one timestamp. The validity interval of the query result must contain the timestamp of the pinned snapshot at which it was executed, by definition. That pinned snapshot was chosen by the TxCache library from the transaction's pin set (or it chose $\star$, obtained a new snapshot, and added it to the pin set). Thus, at least that one timestamp will remain in the pin set after intersecting it with the query's validity interval.

6.3 Handling Nested Calls

In the preceding sections, we assumed that cacheable functions never call other cacheable functions. However, it is useful to be able to nest calls to cacheable functions. For example, a user's home page at an auction site might contain a list of items the user recently bid on. We might want to cache the description and price for each item as a function of the item ID (because they might appear on other user's pages) in addition to the complete content of the user's page (because he might access it again).

Our implementation supports nested calls; this does not require any fundamental changes to the approach above. However, we must keep track of a separate cumulative validity interval and invalidation tag set for each cacheable function in the call stack. When a cached value or database query result is accessed, its validity interval is intersected with that of *each* function currently on the call stack. As a result, a nested call to a cacheable function may have a wider validity interval than its enclosing function, but not vice versa. This makes sense, as the outer function might have seen more data than the functions it calls (*e.g.* if it calls more than one cacheable function). Similarly, any invalidation tags from the database are attached to each function on the call stack, as each now has a dependency on the data.

7 Experiences

We implemented all the components of TxCache, including the cache server, database modifications to PostgreSQL to support validity tracking and invalidations, and the cache library with PHP language bindings.

One of TxCache's goals is to make it easier to add caching to a new or existing application. The TxCache library makes it straightforward to designate a function as cacheable. However, ensuring that the program has functions suitable for caching still requires some effort. Below, we describe our experiences adding support for caching to the RUBiS benchmark and to MediaWiki.

7.1 Porting RUBiS

RUBiS [2] is a benchmark that implements an auction website modeled after eBay where users can register items for sale, browse listings, and place bids on items. We ported its PHP implementation to use TxCache. Like many small PHP applications, the PHP implementation of RUBiS consists of 26 separate PHP scripts, written in an unstructured way, which mainly make database queries and format their output. Besides changing code that begins and ends transactions to use TxCache's interfaces, porting RUBiS to TxCache involved identifying and designating cacheable functions. The existing implementation had few functions, so we had to begin by dividing it into functions; this was not difficult and would be unnecessary in a more modular implementation.

We cached objects at two granularities. First, we cached large portions of the generated HTML output (except some headers and footers) for each page. This meant that if two clients viewed the same page with the same arguments, the previous result could be reused. Second, we cached common functions such as authenticating a user's login, or looking up information about a user or item by ID. Even these fine-grained functions were often more complicated than an individual query; for example, looking up an item requires examining both the active

items table and the old items table. These fine-grained cached values can be shared between different pages; for example, if two search results contain the same item, the description and price of that item can be reused.

We made a few modifications to RUBiS that were not strictly necessary but improved its performance. To take better advantage of the cache, we modified the code for display lists of items to obtain details about each item by calling our GET-ITEM cacheable function rather than performing a join on the database. We also observed that one interaction, finding all the items for sale in a particular region and category, required performing a sequential scan over all active auctions, and joining it against the users table. This severely impacted the performance of the benchmark with or without caching. We addressed this by adding a new table and index containing each item's category and region IDs. Finally, we removed a few queries that were simply redundant.

7.2 Porting MediaWiki

We also ported MediaWiki to use TxCache, to better understand the process of adding caching to a more complex, existing system. MediaWiki, which faces significant scaling challenges in its use for Wikipedia, already supports a variety of caches and replication systems. Unlike RUBiS, it has an object-oriented design, making it easier to select cacheable functions.

MediaWiki supports master-slave replication for the database server. Because the slaves cannot process update transactions and lag slightly behind the master, MediaWiki already distinguishes the few transactions that must see the latest state from the majority that can accept the staleness caused by replication lag (typically 1–30 seconds). It also identifies read/write transactions, which must run on the master. Although we used only one database server, we took advantage of this classification of transactions to determine which transactions can be cached and which must execute directly on the database.

Most MediaWiki functions are class member functions. Caching only pure functions requires being sure that functions do not mutate their object. We cached only static functions that do not access or modify global variables (MediaWiki rarely uses global variables). Of the non-static functions, many can be made static by explicitly passing in any member variables that are used, as long as they are only read. For example, almost every function in the TITLE class, which represents article titles, is cacheable because a TITLE object is immutable.

Identifying functions that would be good candidates for caching was more challenging, as MediaWiki is a complex application with myriad features. Developers with previous experience with the MediaWiki codebase would have more insight into which functions were frequently used. We looked for functions that were involved in common requests like rendering an article, and member functions of commonly-used classes. We focused on functions that constructed objects based on data looked up in the database, such as fetching a page revision. These were good candidates for caching because we can avoid the cost of one or more database queries, as well as the cost of post-processing the data from the database to fill the fields of the object. We also adapted existing caches like the localization cache, which stores translations of user interface messages.

8 Evaluation

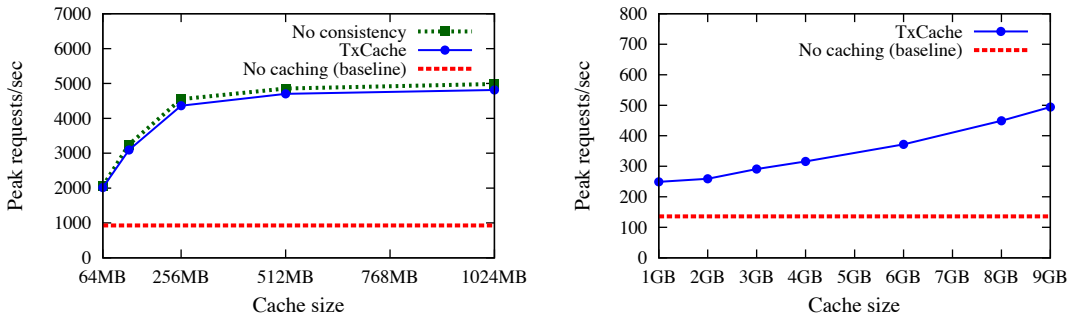
We used RUBiS as a benchmark to explore the performance benefits of caching. In addition to the PHP auction site implementation described above, RUBiS provides a client emulator that simulates many concurrent user sessions: there are 26 possible user interactions (*e.g.* browsing items by category, viewing an item, or placing a bid), each of which corresponds to a transaction. We used the standard RUBiS "bidding" workload, a mix of 85% read-only interactions (browsing) and 15% read/write interactions (placing bids) with a think time with negative exponential distribution and 7-second mean.

We ran our experiments on a cluster of 10 servers, each a Dell PowerEdge SC1420 with two 3.20 GHz Intel Xeon CPUs, 2 GB RAM, and a Seagate ST31500341AS 7200 RPM hard drive. The servers were connected via a gigabit Ethernet switch, with 0.1 ms round-trip latency. One server was dedicated to the database; it ran PostgreSQL 8.2.11 with our modifications. The others acted as front-end web servers running Apache 2.2.12 with PHP 5.2.10, or as cache nodes. Four other machines, connected via the same switch, served as client emulators. Except as otherwise noted, database server load was the bottleneck.

We used two different database configurations. One configuration was chosen so that the dataset would fit easily in the server's buffer cache, representative of applications that strive to fit their working set into the buffer cache for performance. This configuration had about 35,000 active auctions, 50,000 completed auctions, and 160,000 registered users, for a total database size about 850 MB. The larger configuration was disk-bound; it had 225,000 active auctions, 1 million completed auctions, and 1.35 million users, for a total database size of 6 GB.

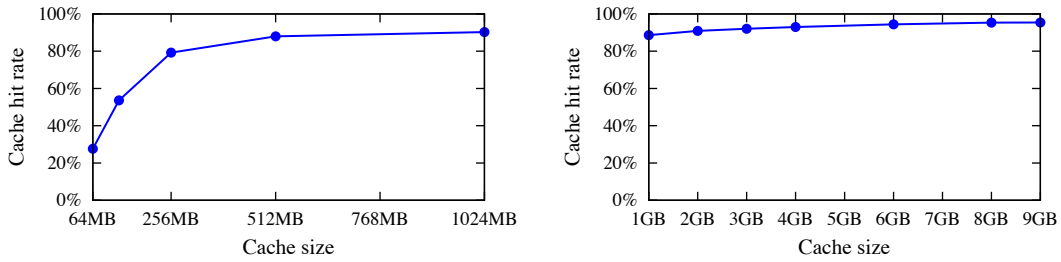
For repeatability, each test ran on an identical copy of the database. We ensured the cache was warm by restoring its contents from a snapshot taken after one hour of continuous processing for the in-memory configuration and one day for the disk-bound configuration.

For the in-memory configuration, we used seven hosts as web servers, and two as dedicated cache nodes. For the larger configuration, eight hosts ran both a web server and a cache server, in order to make a larger cache available.



(a) In-memory database (b) Disk-bound database

Figure 5: Effect of cache size on peak throughput (30 second staleness limit)



(a) In-memory database (b) Disk-bound database

Figure 6: Effect of cache size on cache hit rate (30 second staleness limit)

8.1 Cache Sizes and Performance

We evaluated RUBiS's performance in terms of the peak throughput achieved (requests handled per second) as we varied the number of emulated clients. Our baseline measurement evaluates RUBiS running directly on the Postgres database, with TxCache disabled. This achieved a peak throughput of 928 req/s with the in-memory configuration and 136 req/s with the disk-bound configuration.

We performed this experiment with both a stock copy of Postgres, and our modified version. We found no observable difference between the two cases, suggesting our modifications have negligible performance impact. Because the system already maintains multiple versions to implement snapshot isolation, keeping a few more versions around adds little cost, and tracking validity intervals and invalidation tags simply adds an additional bookkeeping step during query execution.

We then ran the same experiment with TxCache enabled, using a 30 second staleness limit and various cache sizes. The resulting peak throughput levels are shown in Figure 5. Depending on the cache size, the speedup achieved ranged from 2.2× to 5.2× for the in-memory configuration and from 1.8× to 3.2× for the disk-bound configuration. The RUBiS PHP benchmark does not perform significant application-level computation; even so, we see a 15% reduction in total web server CPU usage. Cache server load is low, with most CPU overhead in kernel time, suggesting inefficiencies in the kernel's TCP stack as the cause. Switching to a UDP protocol might alleviate some of this overhead [32].

Figure 6(a) shows that for the in-memory configuration, the cache hit rate ranged from 27% to 90%, increasing linearly until the working set size is reached, and then growing slowly. Here, the cache hit rate directly translates into a performance improvement because each cache hit represents load (often many queries) removed from the database. Interestingly, we always see a high hit rate on the disk-bound database (Figure 6(b)) but it does not always translate into a large performance improvement. This workload exhibits some very frequent queries (*e.g.* looking up a user's nickname by ID) that can be stored in even a small cache, but are also likely to be in the database's buffer cache. It also has a large number of data items that are each accessed rarely (*e.g.* the full bid history for each item). The latter queries collectively make up the bottleneck, and the speedup is determined by how much of this data is in the cache.

8.2 Varying Staleness Limits

The staleness limit is an important parameter. By raising this value, applications may be exposed to increasingly stale data, but are able to take advantage of more cached

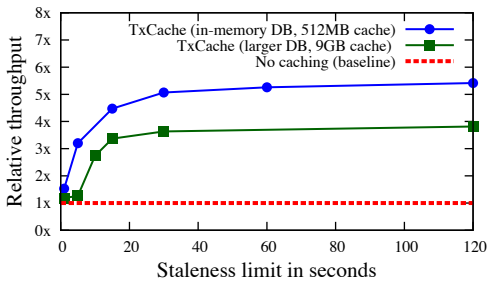


Figure 7: Impact of staleness limit on peak throughput

data. An invalidated cache entry remains useful for the duration of the staleness limit, which is valuable for values that change (and are invalidated) frequently.

Figure 7 compares the peak throughput obtained by running transactions with staleness limits from 1 to 120 seconds. Even a small staleness limit of 5-10 seconds provides a significant benefit. RUBiS has some objects that are expensive to compute and have many data dependencies (indexes of all items in particular regions with their current prices). These objects are invalidated frequently, but the staleness limit permits them to be used. The benefit diminishes at around 30 seconds, suggesting that the bulk of the data either changes infrequently (such as information about inactive users or auctions), or is accessed multiple times every 30 seconds (such as the aforementioned index pages).

8.3 Costs of Consistency

A natural question is how TxCache's guarantee of transactional consistency affects its performance. We explore this question by examining cache statistics and comparing against other approaches.

We classified cache misses into four types, inspired by the common classification for CPU cache misses:

- *compulsory miss*: the object was never in the cache
- *staleness miss*: the object has been invalidated, and its staleness limit has been exceeded
- *capacity miss*: the object was previously evicted
- *consistency miss*: some sufficiently fresh version of the object was available, but it was inconsistent with previous data read by the transaction

Figure 8 shows the breakdown of misses by type for four different configurations. Our cache server unfortunately cannot distinguish staleness and capacity misses. We see that consistency misses are the least common by a large margin. Consistency misses are rare, as items in the cache are likely to have overlapping validity intervals, either because they change rarely or the cache contains multiple versions. Workloads with higher staleness limits experience more consistency misses (but fewer overall misses) because they have more stale data that must be matched

	in-memory DB			disk-bound
	512 MB	512 MB	64 MB	9 GB
	30 s stale	15 s stale	30 s stale	30 s stale
Compulsory	33.2%	28.5%	4.3%	63.0%
Stale / Cap.	59.0%	66.1%	95.5%	36.3%
Consistency	7.8%	5.4%	0.2%	0.7%

Figure 8: Breakdown of cache misses by type. Figures are percentage of total misses.

to other items valid at the same time. The 64 MB-sized cache's workload is dominated by capacity misses, because the cache is smaller than the working set. The disk-bound experiment sees more compulsory misses because it has a larger dataset with limited locality, and few consistency misses because the update rate is slower.

The low fraction of consistency misses suggests that providing consistency has little performance cost. We verified this experimentally by modifying our cache to continue to use our invalidation mechanism, but to read any data that was valid within the last 30 seconds, blithely ignoring consistency. The results of this experiment are shown as the "No consistency" line in Figure 5(a). As predicted, the benefit it provides over consistency is small. On the disk-bound configuration, the results could not be distinguished within experimental error.

9 Related Work

High performance web applications use many different techniques to improve their throughput. These range from lightweight application-level caches which typically do not provide transactional consistency, to database replication systems that improve database performance while providing the same consistency guarantees, but do not address application server load.

9.1 Application-Level Caching

Applying caching at the application layer is an appealing option because it can improve performance of both the application servers and the database. Dynamic web caches operate at the highest layer, storing entire web pages produced by the application, requiring them to be regenerated in their entirety when any content changes. These caches need to invalidate pages when the underlying data changes, typically by requiring the application to explicitly invalidate pages [37] or specify data dependencies [9, 38]. TxCache obviates this need by integrating with the database to automatically identify dependencies.

However, full-page caching is becoming less appealing to application developers as more of the web becomes personalized and dynamic. Instead, web developers are increasingly turning to application-level data caches [4, 16, 24, 26, 34] for their flexibility. These caches allow the application to choose what to store, including query results, arbitrary application data (such as Java or .NET

objects), and fragments of or whole web pages.

These caches present to applications a GET/PUT/DELETE hash table interface, so the application developer must choose keys and correctly invalidate objects. As we argued in Section 2.1, this can be a source of unnecessary complexity and software bugs. Most application object caches have no notion of transactions, so they cannot ensure even that two accesses to the cache return consistent values. Some support transactions *within* the cache, allowing applications to atomically update objects in the cache [34, 16], but none maintain transactional consistency with the database.

9.2 Database Replication

Another popular alternative is to deploy a caching or replication system within the database layer. These systems replicate the data tuples that comprise the database, and allow replicas to perform queries on them. Accordingly, they can relieve load on the database, but offer no benefit for application server load.

Some replication systems guarantee transactional consistency by using group communication to execute queries [12, 19], which can be difficult to scale to large numbers of replicas [13]. Others offer weaker guarantees (eventual consistency) [11, 27], which can be difficult to reason about and use correctly. Still others require the developer to know the access pattern beforehand [3] or statically partition the data [8].

Most replication schemes used in practice take a primary copy approach, where all modifications are processed at a master and shipped to slave replicas, usually asynchronously for performance reasons. Each replica then maintains a complete, if slightly stale, copy of the database. Several systems defer update processing to improve performance for applications that can tolerate limited amounts of staleness [6, 28, 30]. These protocols assume that each replica is a single, complete snapshot of the database, making them infeasible for use in an application object cache setting where it is not possible to maintain a copy of every object that could be computed. In contrast, TxCache's protocol allows it to ensure consistency even though its cache contains cached objects that were generated at different times.

Materialized views are a form of in-database caching that creates a view table containing the result of a query over one or more base tables, and updating it as the base tables change. Most work on materialized views seeks to incrementally update the view rather than recomputing it in its entirety [15]. This requires placing restrictions on view definitions, *e.g.* requiring them to be expressed in the select-project-join algebra. TxCache's application-level functions, in addition to being computed outside the database, can include arbitrary computation, making incremental updates infeasible. Instead, it uses invalidations, which are easier for the database to compute [7].

10 Conclusion

Application data caches are an efficient way to scale database-driven web applications, but they do not integrate well with databases or web applications. They break the consistency guarantees of the underlying database, making it impossible for the application to see a consistent view of the entire system. They provide a minimal interface that requires the application to provide significant logic for keeping cached values up to date, and often requires application developers to understand the entire system in order to correctly manage the cache.

We provide an alternative with TxCache, an application-level cache that ensures all data seen by an application during a transaction is consistent, regardless of whether it comes from the cache or database. TxCache guarantees consistency by modifying the database server to return validity intervals, tagging data in the cache with these intervals, and then only retrieving values from the cache that were valid at a single point in time. By using validity intervals instead of single timestamps, TxCache can make the best use of cached data by lazily selecting the timestamp for each transaction.

TxCache provides an easier programming model for application developers by allowing them to simply designate cacheable functions, and then have the results of those functions automatically cached. The TxCache library handles all of the complexity of managing the cache and maintaining consistency across the system: it selects keys, finds data in the cache consistent with the current transaction, and automatically detects and invalidates potentially changed objects as the database is updated.

Our experiments with the RUBiS benchmark show that TxCache is effective at improving scalability even when the application tolerates only a small interval of staleness, and that providing transactional consistency imposes only a minor performance penalty.

Acknowledgments

We thank James Cowling, Kevin Grittner, our shepherd Amin Vahdat, and the anonymous reviewers for their helpful feedback. This research was supported by NSF ITR grants CNS-0428107 and CNS-0834239, and by NDSEG and NSF graduate fellowships.

References

[1] C. Amza, E. Cecchet, A. Chanda, S. Elnikety, A. Cox, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Bottleneck characterization of dynamic web site benchmarks. TR02-388, Rice University, 2002.

[2] C. Amza, A. Chanda, A. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic web site

benchmarks. *Proc. Workshop on Workload Characterization*, Nov. 2002.

[3] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: consistent replication for scaling back-end databases of dynamic content web sites. In *Proc. Middleware '03*, Rio de Janeiro, Brazil, June 2003.

[4] R. Bakalova, A. Chow, C. Fricano, P. Jain, N. Kodali, D. Poirier, S. Sankaran, and D. Shupp. WebSphere dynamic cache: Improving J2EE application experience. *IBM Systems Journal*, 43(2), 2004.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. SIGMOD '95*, San Jose, CA, June 1995.

[6] P. A. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma. Relaxed-currency serializability for middle-tier caching and replication. In *Proc. SIGMOD '06*, Chicago, IL, 2006.

[7] K. S. Candan, D. Agrawal, W.-S. Li, O. Po, and W.-P. Hsiung. View invalidation for dynamic content caching in multitiered architectures. In *Proc. VLDB '02*, Hong Kong, China, 2002.

[8] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: flexible database clustering middleware. In *Proc. USENIX '04*, Boston, MA, June 2004.

[9] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *Proc. INFOCOM '99*, Mar 1999.

[10] J. Cowling, D. R. K. Ports, B. Liskov, R. A. Popa, and A. Gaikwad. Census: Location-aware membership management for large-scale distributed systems. In *Proc. USENIX '09*, San Diego, CA, June 2009.

[11] A. Downing, I. Greenberg, and J. Peha. OSCAR: a system for weak-consistency replication. In *Proc. Workshop on Management of Replicated Data*, Nov 1990.

[12] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *Proc. SRDS '05*, Washington, DC, 2005.

[13] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. SIGMOD '96*, Montreal, QC, June 1996.

[14] P. J. Guo and D. Engler. Towards practical incremental recomputation for scientists: An implementation for the Python language. In *Proc. TAPP '10*, San Jose, CA, Feb. 2010.

[15] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. SIGMOD '93*, Washington, DC, June 1993.

[16] JBoss Cache. http://www.jboss.org/jbosscache/.

[17] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. STOC '97*, El Paso, TX, May 1997.

[18] K. Keeton, C. B. Morrey III, C. A. N. Soules, and A. Veitch. LazyBase: Freshness vs. performance in information management. In *Proc. HotStorage '10*, Big Sky, MT, Oct. 2009.

[19] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *Transactions on Database Systems*, 25(3):333–379, 2000.

[20] L. Lamport. Time, clocks, and ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[21] B. Liskov and R. Rodrigues. Transactional file systems can be fast. In *Proc. ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.

[22] MediaWiki bugs. http://bugzilla.wikimedia.org/. Bugs #7474, #7541, #7728, #10463.

[23] MediaWiki bugs. http://bugzilla.wikimedia.org/. Bugs #8391, #17636.

[24] memcached: a distributed memory object caching system. http://www.danga.com/memcached.

[25] NCache. http://www.alachisoft.com/ncache/.

[26] OracleAS web cache. http://www.oracle.com/technology/products/ias/web_cache/.

[27] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. SOSP '97*, Saint Malo, France, 1997.

[28] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Proc. Middleware '05*, Toronto, Canada, Nov. 2004.

[29] PostgreSQL. http://www.postgresql.org/.

[30] U. Röhm, K. Böhm, H. Schek, and H. Schuldt. FAS: a freshness-sensitive coordination middleware for a cluster of OLAP components. In *Proc. VLDB '02*, Hong Kong, China, 2002.

[31] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware '01*, Heidelberg, Germany, Nov. 2001.

[32] P. Saab. Scaling memcached at Facebook. http://www.facebook.com/note.php?note_id=39391378919, Dec. 2008.

[33] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *Proc. VMCAI '05*, Paris, France, Jan. 2005.

[34] N. Sampathkumar, M. Krishnaprasad, and A. Nori. Introduction to caching with Windows Server AppFabric. Technical report, Microsoft Corporation, Nov 2009.

[35] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *Transactions on Networking*, 11(1):149–160, Feb. 2003.

[36] M. Stonebraker. The design of the POSTGRES storage system. In *Proc. VLDB '87*, Brighton, United Kingdom, Sept. 1987.

[37] H. Yu, L. Breslau, and S. Shenker. A scalable web cache consistency architecture. *SIGCOMM Comput. Commun. Rev.*, 29(4):163–174, 1999.

[38] H. Zhu and T. Yang. Class-based cache management for dynamic web content. In *Proc. INFOCOM '01*, 2001.

Piccolo: Building Fast, Distributed Programs with Partitioned Tables

Russell Power Jinyang Li

New York University
http://news.cs.nyu.edu/piccolo

Abstract

Piccolo is a new data-centric programming model for writing parallel in-memory applications in data centers. Unlike existing data-flow models, Piccolo allows computation running on different machines to share distributed, mutable state via a key-value table interface. Piccolo enables efficient application implementations. In particular, applications can specify locality policies to exploit the locality of shared state access and Piccolo's run-time automatically resolves write-write conflicts using user-defined accumulation functions.

Using Piccolo, we have implemented applications for several problem domains, including the PageRank algorithm, *k*-means clustering and a distributed crawler. Experiments using 100 Amazon EC2 instances and a 12 machine cluster show Piccolo to be faster than existing data flow models for many problems, while providing similar fault-tolerance guarantees and a convenient programming interface.

1 Introduction

With the increased availability of data centers and cloud platforms, programmers from different problem domains face the task of writing parallel applications that run across many nodes. These application range from machine learning problems (k-means clustering, neural networks training), graph algorithms (PageRank), scientific computation etc. Many of these applications extensively access and mutate shared intermediate state stored in memory.

It is difficult to parallelize in-memory computation across many machines. As the entire computation is divided among multiple threads running on different machines, one needs to coordinate these threads and share intermediate results among them. For example, to compute the PageRank score of web page *p*, a thread needs to access the PageRank scores of *p*'s "neighboring" web pages, which may reside in the memory of threads running on different machines. Traditionally, parallel in-memory applications have been built using message-passing primitives such as MPI [21]. For many users, the communication-centric model provided by message-passing is too low-level an abstraction - they fundamentally care about data and processing data, as opposed to the location of data and how to get to it.

Data-centric programming models [19, 27, 1], in which users are presented with a simplified interface to access data but no explicit communication mechanism, have proven a convenient and popular mechanism for expressing many computations. MapReduce and Dryad [27] provide a data-flow programming model that does not expose any globally shared state. While the data-flow model is ideally suited for bulk-processing of on-disk data, it is not a natural fit for in-memory computation: applications have no online access to intermediate state and often have to emulate shared memory access by joining multiple data streams. Distributed shared memory [29, 32, 7, 17] and tuple spaces [13] allow sharing of distributed in-memory state. However, their simple memory (or tuple) model makes it difficult for programmers to optimize for good application performance in a distributed environment.

This paper presents Piccolo, a data-centric programming model for writing parallel in-memory applications across many machines. In Piccolo, programmers organize the computation around a series of application kernel functions, where each kernel is launched as multiple instances concurrently executing on many compute nodes. Kernel instances share distributed, mutable state using a set of in-memory tables whose entries reside in the memory of different compute nodes. Kernel instances share state exclusively via the key-value table interface with *get* and *put* primitives. The underlying Piccolo runtime sends messages to read and modify table entries stored in the memory of remote nodes.

By exposing shared global state, the programming model of Piccolo offers several attractive features. First, it allows for natural and efficient implementations for ap-

plications that require sharing of intermediate state such as k-means computation, n-body simulation, PageRank calculation etc. Second, Piccolo enables online applications that require immediate access to modified shared state. For example, a distributed crawler can learn of newly discovered pages quickly as a result of state updates done by ongoing web crawls.

Piccolo borrows ideas from existing data-centric systems to enable efficient application implementations. Piccolo enforces atomic operations on individual key-value pairs and uses user-defined accumulation functions to automatically combine concurrent updates on the same key (similar to reduce functions in MapReduce [19]). The combination of these two techniques eliminates the need for fine-grained application-level synchronization for most applications. Piccolo allows applications to exploit locality of access to shared state. Users control how table entries are partitioned across machines by defining a partitioning function [19]. Based on users' locality policies, the underlying run-time can schedule a kernel instance where its needed table partitions are stored, thereby reducing expensive remote table access.

We have built a run-time system consisting of one master (for coordination) and several worker processes (for storing in-memory table partitions and executing kernels). The run-time uses a simple work stealing heuristic to dynamically balance the load of kernel execution among workers. Piccolo provides a global checkpoint/restore mechanism to recover from machine failures. The run-time uses the Chandy-Lamport snapshot algorithm [15] to periodically generate a consistent snapshots of the execution state without pausing active computations. Upon machine failure, Piccolo recovers by restarting the computation from its latest snapshot state.

Experiments have shown that Piccolo is fast and provides excellent scaling for many applications. The performance of PageRank and k-means on Piccolo is $11\times$ and $4\times$ faster than that of Hadoop. Computing a PageRank iteration for a 1 billion-page web graph takes only 70 seconds on 100 EC2 instances. Our distributed web crawler can easily saturate a 100 Mbps internet uplink when running on 12 machines.

The rest of the paper is organized as follows. Section 2 provides a description of the Piccolo programming model, followed by the design of Piccolo's run-time (Section 3). We describe the set of applications we constructed using Piccolo in Section 4. Section 5 discusses our prototype implementation. We show Piccolo's performance evaluation in Section 6 and present related work in Section 7.

2 Programming Model

Piccolo's programming environment is exposed as a library to existing languages (our current implementation supports C++ and Python) and requires no change to underlying OS or compiler. This section describes the programming model in terms of how to structure application programs (§2.1), share intermediate state via key/value tables (§2.2), optimize for locality of access (§2.3), and recover from failures(§2.4). We conclude this section by showing how to implement the PageRank algorithm on top of Piccolo (§2.5).

2.1 Program structure

Application programs written for Piccolo consist of *control* functions which are executed on a single machine, and *kernel* functions which are executed concurrently on many machines. Control functions create shared tables, launch multiple instances of a kernel function, and perform global synchronization. Kernel functions consist of sequential code which read from and write to tables to share state among concurrently executing kernel instances. By default, control functions execute in a single thread and a single thread is created for executing each kernel instance. However, the programmer is free to create additional application threads in control or kernel functions as needed.

Kernel invocation: The programmer uses the `Run` function to launch a specified number (m) of kernel instances executing the desired kernel function on different machines. Each kernel instance has an identifier $0 \cdots m-1$ which can be retrieved using the `my_instance` function.

Kernel synchronization: The programmer invokes a global barrier from within a control function to wait for the completion of all previously launched kernels. Currently, Piccolo does not support pair-wise synchronization among concurrent kernel instances. We found that global barriers are sufficient because Piccolo's shared table interface makes most fine-grained locking operations unnecessary. This overall application structure, where control functions launch kernels across one or more global barriers, is reminiscent of the CUDA model [36] which also explicitly eschews support for pair-wise thread synchronization.

2.2 Table interface and semantics

Concurrent kernel instances share intermediate state across machine through key-value based in-memory tables. Table entries are spread across all nodes and each key-value pair resides in the memory of a single node. Each table is associated with explicit key and value types which can be arbitrary user-declared serializable types. As Figure 1 shows, the key-value interface provides a uniform access model whether the underlying table en-

```
Table<Key, Value>:
  clear()
  contains(Key)
  get(Key)
  put(Key, Value)

  # updates the existing entry via
  # user-defined accumulation.
  update(Key, Value)

  # Commit any buffered updates/puts
  flush()

  # Return an iterator on a table partition
  get_iterator(Partition)
```

Figure 1: Shared Table Interface

try is stored locally or on another machine. The table APIs include standard operations such as `get`, `put` as well as Piccolo-specific functions like `update`, `flush`, `get_iterator`. Only control functions can create tables; both control and kernel functions can invoke any table operation.

User-defined accumulation: Multiple kernel instances can issue concurrent `update`s to the same key. To resolve such write-write conflict, the programmer can associate a user-defined accumulation function with each table. Piccolo executes the accumulator during run-time to combine concurrent `update`s on the same key. If the programmer expects results to be independent from the ordering of updates, the accumulator must be a commutative and associative function [52].

Piccolo provides a set of standard accumulators such as summation, multiplication and min/max. To define an accumulator, the user specifies four functions: `Initialize` to initialize an accumulator for a newly created key, `Accumulate` to incorporate the effect of a single `update` operation, `Merge` to combine the contents of multiple accumulators on the same key, and `View` to return the current accumulator state reflecting all `update`s accumulated so far. Accumulator functions have no access to global state except for the corresponding table entry being updated.

User-controlled Table Partitioning: Piccolo uses a user-specified *partition function* [19] to divide the key-space into partitions. Table partitioning is a key primitive for expressing user programs' locality preferences. The programmer specifies the number of partitions (p) when creating a table. The p partitions of a table are named with integers $0...p-1$. Kernel functions can scan all entries in a given table partition using the `get_iterator` function (see Figure 1).

Piccolo does not reveal to the programmer which node stores a table partition, but guarantees that all table entries in a given partition are stored on the same machine. Although the run-time aims to have a load-balanced assignment of table partitions to machines, it is the programmer's responsibility to ensure that the largest table partition fits in the available memory of a single machine. This can usually be achieved by specifying a the number of partitions to be much larger than the number of machines.

Table Semantics: All table operations involving a single key-value pair are atomic from the application's perspective. Write operations (e.g. `update`, `put`) destined for another machine can be buffered to avoid blocking kernel execution. In the face of buffered remote writes, Piccolo provides the following guarantees:

- All operations issued by a single kernel instance on the same key are applied in their issuing order. Operations issued by different kernel instances on the same key are applied in some total order [31].
- Upon a successful `flush`, all buffered writes done by the caller's kernel instance will have been committed to their respective remote locations, and will be reflected in the response to subsequent `get`s by any kernel instance.
- Upon the completion of a global barrier, all kernel instances will have been completed and all their writes will have been applied.

2.3 Expressing locality preferences

While writes to remote table entries can be buffered at the local node, the communication latency involved in fetching remote table entries cannot be effectively hidden. Therefore, the key to achieving good application performance is to minimize remote `get`s by exploiting locality of access. By organizing the computation as kernels and shared state as partitioned tables, Piccolo provides a simple way for programmers to express locality policies. Such policies enable the underlying Piccolo run-time to execute a kernel instance on a machine that stores most of its needed data, thus minimizing remote reads.

Piccolo supports two kinds of locality policies: (1) co-locate a kernel execution with some table partition, and (2) co-locate partitions of different tables. When launching some kernel, the programmer can specify a table argument in the `Run` function to express their preference for co-locating the kernel execution with that table. The programmer usually launches the same number of kernel instances as the number of partitions in the specified table. The run-time schedules the i-th kernel instance to execute on the machine that stores the i-th partition of the specified table. To optimize for kernels that read from more than one table, the programmer uses the `GroupTables(T1,T2,..)` function to co-locate multiple tables. The run-time assigns the i-th partition of T1,T2,...

to be stored on the same machine. As a result, by co-locating kernel execution with one of the tables, the programmer can avoid remote reads for kernels that read from the same partition of multiple tables.

2.4 User-assisted checkpoint and restore

Piccolo handles machine failures via a global checkpoint/restore mechanism. The mechanism currently implemented is not fully automatic - Piccolo saves a consistent global snapshot of all shared table state, but relies on users to save additional information to recover the position of their kernel and control function execution. We believe this design makes a reasonable trade-off. In practice, the programming efforts required for checkpointing user information are relatively small. On the other hand, our design avoids the overhead and complexities involved in automatically checkpointing C/C++ executables.

Based on our experience of writing applications, we arrived at two checkpointing APIs: one synchronous (`CpBarrier`) and one asynchronous (`CpPeriodic`). Both functions are invoked from some control function. Synchronous checkpoints are well-suited for iterative applications (e.g. PageRank) which launch kernels in multiple rounds separated by global barriers and desire to save intermediate state every few rounds. On the other hand, applications with long running kernels (e.g. a distributed crawler) need to use asynchronous checkpoints to save their state periodically.

`CpBarrier` takes as arguments a list of tables and a dictionary of user data to be saved as part of the checkpoint. Typical user data contain the value of some iterator in the control thread. For example in PageRank, the programmer would like to record the number of PageRank iterations computed so far as part of the global checkpoint. `CpBarrier` performs a global barrier and ensures that the checkpointed state is equivalent to the state of execution at the barrier.

`CpPeriodic` takes as arguments a list of tables, a time interval for periodic checkpointing, and a kernel callback function `CheckpointCallback`. This callback is invoked for all active kernels on a node immediately after that node has checkpointed the state for its assigned table partitions. The callback function provides a way for the programmer to save the necessary data required to restore running kernel instances. Oftentimes this is the position of an iterator over the partition that is being processed by a kernel instance. When restoring, Piccolo reloads the table state on all nodes, and invokes kernel instances with the dictionary saved during the checkpoint.

2.5 Putting it together: PageRank

As a concrete example, we show how to implement PageRank using Piccolo. The PageRank algorithm [11] takes as input a sparse web graph and computes a rank value for each page. The computation proceeds in multiple iterations: page i's rank value in the k-th iteration ($p_i^{(k)}$) is the sum of the normalized ranks of its incoming neighbors in the previous iteration, i.e. $p_i^{(k)} = \sum_{\forall j \in In_i} \frac{p_j^{(k-1)}}{|Out_j|}$, where Out_j denotes page j's outgoing neighbors.

```
tuple PageID(site, page)
const PropagationFactor = 0.85

def PRKernel(Table(PageID,double) curr,
             Table(PageID,double) next,
             Table(PageID,[PageID]) graph_partition):
  for page, outlinks in
      graph.get_iterator(my_instance()):
    rank = curr[page]
    update = PropagationFactor * rank / len(outlinks)
    for target in outlinks:
      next.update(target, update)

def PageRank(Config conf):
  graph = Table(PageID,[PageID]).init("/dfs/graph")
  curr = Table(PageID, double).init(
        graph.numPartitions(),
        SumAccumulator, SitePartitioner)

  next = Table(PageID, double).init(
        graph.numPartitions(),
        SumAccumulator, SitePartitioner)

  GroupTables(curr, next, graph)

  if conf.restore():
    last_iter = curr.restore_from_checkpoint()
  else: last_iter = 0

  # run 50 iterations
  for i in range(last_iter, 50):
    Run(PRKernel,
        instances=curr_pr.numPartitions(),
        locality=LOC_REQUIRED(curr),
        args=(curr, next, graph))

    # checkpoint every 5 iterations, storing the
    # current iteration alongside checkpoint data
    if i % 5 == 0:
      CpBarrier(tables=curr,
                {iteration=i})
    else: Barrier()

    # the values accumulated into 'next' become the
    # source values for the next iteration
    swap(curr,next)
```

Figure 2: PageRank Implementation

The complete PageRank implementation in Piccolo is shown in Figure 2. The input web graph is represented as a set of outgoing links, $page \rightarrow target$, for each page. The graph is loaded into the shared in-memory table (`graph`) from a distributed file system. For link graphs too large to fit in memory, Piccolo also supports a read-only `DiskTable` interface for streaming data from disk.

The intermediate rank values are kept in two tables: `curr` for the ranks to be read in the current iteration, `next` for the ranks to be written. The control function

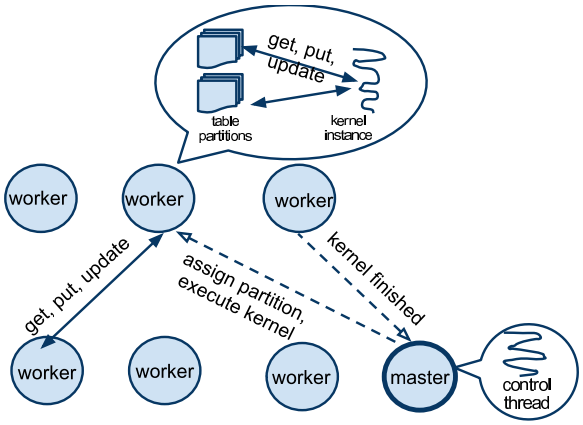


Figure 3: The interactions between master and workers in executing a Piccolo program.

(PageRank) iteratively launches p PRKernel kernel instances where p is the number of table partitions in graph (which is identical to that of curr and next). The kernel instance i scans all pages in the i-th partition of graph. For each $page \rightarrow target$ link, the kernel instance reads the rank value of *page* in curr, and generates updates for next to increment *target*'s rank value for the next iteration.

Since the program generates concurrent updates to the same key in next, it associates the Sum accumulator with next, which correctly combines updates as desired by the PageRank computation. The overall computation proceeds in rounds using a global barrier between PRKernel invocations.

To optimize for locality, the program groups tables graph, curr, next together and expresses preference for co-locating PRKernel executions with the curr table. As a result, none of the kernel instances need to perform any remote reads. In addition, the program uses the partition function, SitePartitioner, to assign the URLs in the same domain to the same partition. As pages in the same domain tend to link to one another frequently, such partitioning significantly reduces the number of remote updates.

Checkpointing/restoration is straightforward: the control thread performs a synchronous checkpoint to save the next table every five iterations and loads the latest checkpointed table to recover from failure.

3 System Design

This section describes the run-time design for executing Piccolo programs on a large collection of machines connected via high-speed Ethernet.

3.1 Overview

Piccolo's execution environment consists of one *master* process and many *worker* processes, each executing on a potentially different machine. Figure 3 illustrates the overall interactions among workers and the master when executing a Piccolo program. As Figure 3 shows, the master executes the user control thread by itself and schedules kernel instances to execute on workers. Additionally, the master decides how table partitions are assigned to workers. Each worker is responsible for storing assigned table partitions in its memory and handling table operations associated with those partitions. Having a single master does not introduce a performance bottleneck: the master informs all workers of the current partition assignment so that workers need not consult the master to perform performance-critical table operations.

The master begins the execution of a Piccolo program by invoking the entry function in the control thread. Upon each table creation API call, the master decides on a partition assignment. The master informs all workers of the partition assignment and each worker initializes its set of partitions, which are all empty at startup. Upon each Run API call to execute m kernel instances, the master prepares m tasks, one for each kernel instance. The master schedules these tasks for execution on workers based on user's locality preferences. Each worker runs a single kernel instance at a time and notifies the master upon task completion. The master instructs each completed worker to proceed with an additional task if it is available. Upon encountering a global barrier, the master blocks the control thread until all active tasks are finished.

During kernel execution, a worker buffers update operations destined for remote workers, combines them using user-defined accumulators and flushes them to remote workers after a short timeout. To handle a get or put operation, the worker flushes accumulated updates on the same key before sending the operation to the remote worker. Each owner applies operations (including accumulated updates) in their received order. Piccolo does not perform caching but supports a limited form of pre-fetching: after each get_iterator API call, the worker pre-fetches a portion of table entries beyond the current iterator value.

Two main challenges arise in the above basic design. First, how to assign tasks in a load-balanced fashion so as to reduce the overall wait time on global barriers? This is particularly important for iterative applications that incur a global barrier at each iteration of the computation. The second challenge is to perform efficient checkpointing and restoration of table state. In the rest of this Section, we detail how Piccolo addresses both challenges.

3.2 Load-balanced Task Scheduling

Basic scheduling without load-balancing works as follows. At table creation time, the master assigns table partitions to all workers using a simple round-robin assign-

ment for empty memory tables. For tables loaded from a distributed file, the master chooses an assignment that minimizes inter-rack transfer while keeping the number of partitions roughly balanced among workers. The master schedules m tasks according to the specified locality preference, namely, it assigns task i to execute on a worker storing partition i.

This initial schedule may not be ideal. Due to heterogeneous hardware configurations or variable-sized computation inputs, workers can take varying amounts of time to finish assigned tasks, resulting in load imbalance and non-optimal use of machines. Therefore, the runtime needs to load-balance kernel executions beyond the initial schedule.

Piccolo's scheduling freedom is limited by two constraints: First, no running tasks should be killed. As a running kernel instance modifies shared table state, re-executing a terminated kernel instance requires performing an expensive restore operation from a saved checkpoint. Therefore, once a kernel instance is started, it is better to let the task complete than terminating it halfway for re-scheduling. By contrast, MapReduce systems do not have this constraint [28] as reducers do not start aggregation until all mappers are finished. The second constraint comes from the need to honor user locality preferences. Specifically, if a kernel instance is to be moved from one worker to another, its co-located table partitions must also be transferred across those workers.

Load-balance via work stealing: Piccolo performs a simple form of load-balancing: the master observes the progress of different workers and instructs a worker (w_{idle}) that has finished all its assigned tasks to steal a not-yet-started task i from the worker (w_{busy}) with the most remaining tasks. We adopt the greedy heuristic of scheduling larger tasks first. To implement this heuristic, the master estimates the input size of each task by the number of keys in its corresponding table partition. The master collects partition size information from all workers at table loading time as well as at each global barrier. The master instructs each worker to execute its assigned tasks in decreasing order of estimated task sizes. Additionally, the idle worker w_{idle} always steals the biggest task among w_{busy}'s remaining tasks.

Table partition migration: Because of user locality preference, worker w_{idle} needs to transfer the corresponding table partition i from w_{busy} before it executes stolen task i. Since table migration occurs while other active tasks are sending operations to partition i, Piccolo must take care not to lose, re-order or duplicate operations from any worker on a given key in order to preserve table semantics. Piccolo uses a multi-phase migration process that does not require suspending any active tasks.

The master coordinates the process of migrating partition i from w_a to w_b, which proceeds in two phases. In the first phase, the master sends message M_1 to all workers indicating the new ownership of i. Upon receiving M_1, all workers flush their buffered operations for i to w_a and begin to send subsequent requests for i to w_b. Upon the receipt of M_1, w_a "pauses" updates to i, and begins to forward requests received from other workers for i to w_b. w_a then transfers the paused state for i to w_b. During this phase, worker w_b buffers all requests for i received from w_a or other workers but does not yet handle them.

After the master has received acknowledgments from all workers that the first phase is complete, it sends M_2 to w_a and w_b to complete migration. Upon receiving M_2, w_a flushes any pending operations destined for i to w_b and discards the paused state for partition i. w_b first handles buffered operations received from w_a in order and then resumes normal operation on partition i.

As can be seen, the migration process does not block any update operations and thus incurs little latency overhead for most kernels. The normal checkpoint/recovery mechanism is used to cope with faults that might occur during migration.

3.3 Fault Tolerance

Piccolo relies on user-assisted checkpoint and restore to cope with both master and worker failures during program execution. The Piccolo run-time saves a checkpoint of program state (including tables and other user-data) on a distributed file system and restores from the latest completed checkpoint to recover from a failure.

Checkpoint: Piccolo needs to save a consistent global checkpoint with low overhead. To ensure consistency, Piccolo must determine a global snapshot of the program state. To reduce overhead, the run-time must carry out checkpointing in the face of actively running kernel instances or the control thread.

We use the Chandy-Lamport (CL) distributed snapshot algorithm [15] to perform checkpointing. To save a CL snapshot, each process records its own state and two processes incident on a communication channel cooperate to save the channel state. In Piccolo, channel state can be efficiently captured using only table modification messages as kernels communicate with each other exclusively via tables.

To begin a checkpoint, the master chooses a new checkpoint epoch number (E) and sends the start checkpoint message $Start_E$ to all workers. Upon receiving the start message, worker w immediately takes a snapshot of the current state of its responsible table partitions and buffers future table operations (in addition to applying them). Once the table partitions in the snapshot are written to stable storage, w sends the marker message $M_{E,w}$ to all other workers. Worker w then enters a logging state in which it logs all buffered operations to a replay

file. Once w has received markers from all other workers ($M_{E,w'}, \forall w' \neq w$), it writes the replay log to stable storage and sends $Fin_{E,w}$ to the master. The master considers the checkpointing done once it has received $Fin_{E,w}$ from all workers.

For asynchronous checkpoints, the master initiates checkpoints periodically based on a timer. To record user-data consistently with recorded table state, each worker atomically takes a snapshot of table state and invokes the checkpoint callback function to save any additional user state for its currently running kernel instance. Synchronous checkpoints provide the semantics that checkpointed state is equivalent to those immediately after the global barrier. Therefore, for synchronous checkpointing, each worker waits until it has completed all its assigned tasks before sending the checkpoint marker $M_{E,w}$ to all other workers. Furthermore, the master saves user-data in the control thread only after it has received $Fin_{E,w}$ from all workers. There is a trade-off in deciding when to start a synchronous checkpoint. If the master starts the checkpoint too early, e.g. while workers still have many remaining tasks, replay files become unnecessarily large. On the other hand, if the master delays checkpointing until all workers have finished, it misses opportunities to overlap kernel computation with checkpointing. Piccolo uses a heuristic to balance this trade-off: the master begins a synchronous checkpoint as soon as one of the workers has finished all its assigned tasks.

To simplify the design, the master does not initiate checkpointing while there is active table migration and vice-versa.

Restore: Upon detecting any worker failure, the master resets the state of all workers and restores computation from the last completed global checkpoint. Piccolo does not checkpoint the internal state of the master - if the master is restarted, restoration occurs as normal, however, the replacement master is free to choose a different partition assignment and task schedule during restoration.

4 More Applications

In addition to PageRank, we have implemented four other applications: a distributed web crawler, k-means, n-body, matrix multiplication. This section summarizes how Piccolo's programming model enables efficient implementation for these applications.

4.1 Distributed Web Crawler

Apart from iterative computations such as PageRank, Piccolo can be used by applications to distribute and coordinate fine-grained tasks among many machines. To demonstrate this usage, we implemented a distributed web crawler. The basic crawler operation is simple: beginning from a few initial URLs, the crawler repeatedly downloads a page and parses it to discover new URLs to fetch. A practical crawler must also satisfy other important constraints: (1) honor the robots.txt file of each web site, (2) refrain from overwhelming a site by capping fetches to a site at a fixed rate, and (3) avoid repeated fetches of the same URL.

```
#local variables kept by each kernel instance
fetch_pool = Queue()
crawl_output = OutputLog('./crawl.data')

def FetcherThread():
  while 1:
    url = fetch_pool.get()
    txt = download_url(url)
    crawl_output.add(url, txt)

    for l in get_links(txt):
      url_table.update(l, ShouldFetch)
    url_table.update(url, Done)


def CrawlKernel(Table(URL,CrawlState) url_table):
  for i in range(20)
    t = FetcherThread()
    t.start()

  while 1:
    for url, status in url_table.my_partition :
      if status == ShouldFetch
        #omit checking domain in robots table
        #omit checking domain in politeness table
        url_table.update(url, Fetching)
        fetch_pool.add(url)
```

Figure 4: Snippet of the crawler implementation.

Our implementation uses three co-located tables:

- The *url_table* stores the crawling state *ToFetch, Fetching, Blacklisted, Done* for each URL. For each URL p in *ToFetch* state, the crawler fetches the corresponding web page and sets p's state to *Fetching*. After the crawler has finished parsing p and extracting its outgoing links, it sets p's state to *Done*.
- The *politeness* table tracks the last time a page was downloaded for each site.
- The *robots* table stores the processed robots file for each site.

The crawler spawns m kernel instances, one for each machine. Our implementation is done in Python in order to utilize Python's web-related libraries. Figure 4 shows the simplified crawler kernel (omitting details for processing robots.txt and capping per-site download rate). Each kernel scans its local `url_table` partitions to find *ToFetch* URLs and processes them using a pool of helper threads. As all three tables are partitioned according to the `SitePartitioner` function and co-located with each other, a kernel instance can efficiently check for the politeness information and robots entries before downloading a URL. Our implementation uses the max accumulator to resolve write-write conflicts on the same

URL in `url_table` according to *Done* > *Blacklisted* > *Fetching* > *ToFetch*. This allows the simple and elegant operation shown in Figure 4, where kernels rediscovering an already-fetched URL p can request updating p's state to *ToFetch* and still arrive at the correct state for p.

Consistent global checkpointing is important for the crawler's recovery. Without global checkpointing, the recovered crawler may find a page p to be *Done* but does not see any of p's extracted links in the `url_table`, possibly causing those URLs to never be crawled. Our implementation performs asynchronous checkpointing every 10 minutes so that the crawler loses no more than 10 minutes worth of progress due to node failure. Restoring from the last checkpoint can result in some pages being crawled more than once (those lost since the last checkpoint), but the checkpoint mechanism guarantees that no pages will "fall through the cracks."

4.2 Parallel computation

***k*-means.** The k-means algorithm is an iterative computation for grouping n data points into k clusters in a multi-dimensional space. Our implementation stores the assigned centers for data points and the positions of centers in shared tables. Each kernel instance processes a subset of data points to compute new center assignments for those data points and update center positions for the next iteration using the summation accumulator.

***n*-body.** This application simulates the dynamics of a set of particles over many discrete time-steps. We implemented an n-body simulation intended for short distances [43], where particles further than a threshold distance (r) apart are assumed to have no effect on each other. During each time-step, a kernel instance processes a subset of particles: it updates a particle's velocity and position based on its current velocity and the positions of other particles within r distance away. Our implementation uses a partition function to divide space into cubes so that a kernel instance mostly performs local reads in order to retrieve those particles within r distance away.

Matrix multiplication. Computing $C = AB$ where A and B are two large matrices is a common primitive in numerical linear algebra. The input and output matrices are divided into $m \times m$ blocks stored in three tables. Our implementation co-locates tables A, B, C. Each kernel instance processes a partition of table C by computing $C_{i,j} = \sum_{k=1}^{m} A_{i,k} \cdot B_{k,j}$.

5 Implementation

Piccolo has been implemented in C++. We provide both C++ and Python APIs so that users can write kernel and control functions in either C++ or Python. We use SWIG [6] for constructing a Python interface to Piccolo. Our implementation re-uses a number of existing libraries, such as OpenMPI for communication, Google's protocol buffers for object serialization, and LZO for compressing on-disk tables.

All the parallel computations (PageRank, k-means, n-body and matrix multiplication) are implemented using the C++ Piccolo API. The distributed crawler is implemented using the Python API.

6 Evaluation

We tested the performance of Piccolo on the applications described in Section 4. Some applications, such as PageRank and k-means, can also be implemented using the existing data-flow model and we compared the performance of Piccolo with that of Hadoop for these applications.

The highlights of our results are:

- Piccolo is fast. PageRank and k-means are 11× and 4× faster than those on Hadoop. When compared against the results published for DryadLinq [53], in which a PageRank iteration on a 900M page graph were performed in 69 seconds, Piccolo finishes an iteration for a 1B page graph in 70 seconds on EC2, while using 1/5 the number of CPU cores.

- Piccolo scales well. For all applications evaluated, increasing the number of workers shows a nearly linear reduction in the computation time. Our 100-instance EC2 experiment on PageRank also demonstrates good scaling.

- Piccolo can help a non-conventional application like the crawler to achieve good parallel performance. Our crawler, despite being implemented in Python, manages to saturate the Internet bandwidth of our cluster.

6.1 Test Setup

Most experiments were performed using our local cluster of 12 machines: 6 of the machines have 1 quad-core Intel Xeon X3360 (2.83GHz) processor with 4GB memory, the other 6 machines have 2 quad-core Xeon E5520 (2.27GHz) processors with 8GB memory. All machines are connected via a commodity gigabit ethernet switch. Our EC2 experiments involve 100 "large instances" each with 7.5GB memory and 2 "virtual cores" where each virtual core is equivalent to a 2007-era single core 2.5GHz Intel Xeon processor. In all experiments, we created one worker process per core and pinned each worker to use that core.

For scaling experiments, we vary the input size of different applications. Table 5 shows the default and maximum input size used for each application. We generate the web link graph for PageRank based on the statistics of a web graph of 100M pages in UK[9]. Specifically, we

Application	Default input size	Maximum input size
PageRank	100M pages	1B pages
k-means	25M points, 100 clusters	1B points, 100 clusters
n-body	100K points	10M points
Matrix Multiply	edge size = 2500	edge size = 6000

Figure 5: Application input sizes

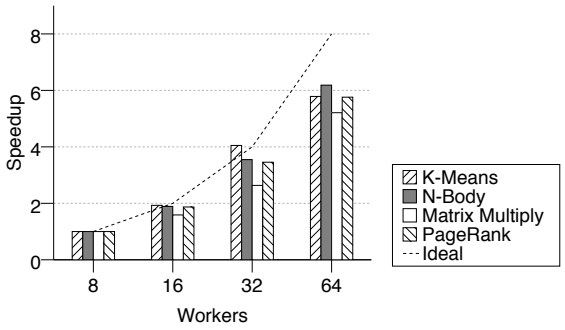


Figure 6: Scaling performance (fixed default input size)

extract the distributions for the number of pages in each site and the ratio of intra/inter-site links. We generate a web graph of any size by sampling from the site size distribution until the desired number of pages is reached; outgoing links are then generated for each page in a site based on the distribution of the ratio of intra/inter-site links. For other applications, we use randomly generated inputs.

6.2 Scaling Performance

Figure 6 shows application speedup as the number of workers (N) increases from 8 to 64 for the default input size. All applications are CPU-bound and exhibit good speedup with increasing N. Ideally, all applications (except for PageRank) have perfectly balanced table partitions and should achieve linear speedup. However, to have reasonable running time at N=8, we choose a relatively small default input size. Thus, as N increases to 64, Piccolo's overhead is no longer negligible relative to applications' own computation (e.g. k-means finishes each iteration in 1.4 seconds at N=64), resulting in 20% less than ideal speedup. PageRank's table partitions are not balanced and work stealing becomes important for its scaling (see § 6.5).

We also evaluate how applications scale with increasing input size by adjusting input size to keep the amount of computation per worker fixed with increasing N. We scale the input size linearly with N for PageRank and k-means. For matrix multiplication, the edge size increases as $O(N^{1/3})$. We do not show results for n-body because it is difficult to scale input size to ensure a fixed amount of computation per worker. For these experiments, the ideal scaling has constant running time as input size increases

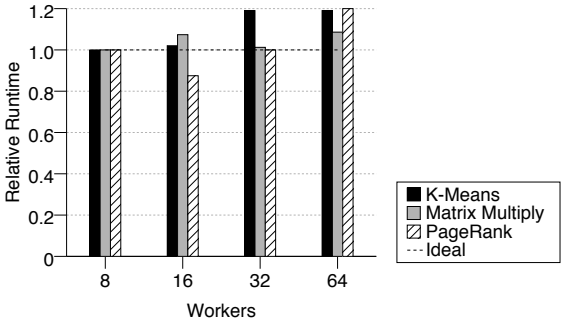


Figure 7: Scaling input size.

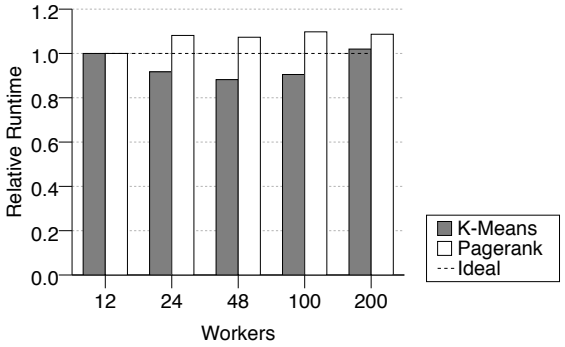


Figure 8: Scaling input size on EC2.

with N. As Figure 7 shows, the achieved scaling for all applications is within 20% of the ideal number.

6.3 EC2

We investigated how Piccolo scales with a larger number of machines using 100 EC2 instances. Figure 8 shows the scaling of PageRank and k-means on EC2 as we increase their input size with N. We were somewhat surprised to see that the resulting scaling on EC2 is better than achieved on our small local testbed. Our local testbed's CPU performance exhibited quite some variability, impacting scaling. After further investigation, we believe the source for such variability is likely due to dynamic CPU frequency scaling.

At N=200, PageRank finishes in 70 seconds for a 1B page link graph. On a similar sized graph (900M pages), our local testbed achieves comparable performance (80 seconds) with many fewer workers (N=64), due to the higher performing cores on our local testbed.

6.4 Comparison with Other Frameworks

Comparison with Hadoop: We implemented PageRank and k-means in Hadoop to compare their performance against that of Piccolo. The rest of our applications, including the distributed web crawler, n-body and matrix

multiplication, do not have any straightforward implementation with Hadoop's data-flow model.

For the Hadoop implementation of PageRank, as with Piccolo, we partition the input link graph by site. During execution, each map task has locality with the partition of graph it is operating on. Mappers join the graph and PageRank score inputs, and use a combiner to aggregate partial results. Our Hadoop *k*-means implementation is highly optimized. Each mapper fetches all 100 centroids from the previous iteration via Hadoop File System (HDFS), computes the cluster assignment of each point in its input stream, and uses a local hash map to aggregate the updates for each cluster. As a result, a reducer only needs to aggregate one update from each mapper to generate the new centroid.

We made extensive efforts to optimize the performance of PageRank and *k*-means on Hadoop including changes to Hadoop itself. Our optimizations include using raw memory comparisons, using primitive types to avoid Java's boxing and unboxing overhead, disabling checksumming, improving Hadoop's join implementation etc. Figure 9 shows the running time of Piccolo and Hadoop using the default input size. Piccolo significantly outperforms Hadoop on both benchmarks (11× for PageRank and 4× for *k*-means with N=64). The performance difference between Hadoop and Piccolo is smaller for *k*-means because of our optimized *k*-means implementation; the structure of PageRank does not admit a similar optimization.

Although we expected to see some performance difference because Hadoop is implemented in Java while Piccolo in C++, the order of magnitude difference came as a surprise. We profiled the PageRank implementation on Hadoop to find the contributing factors. The leading causes for the slowdown are: (1) sorting keys in the map phase (2) serializing and de-serializing data streams and (3) reading and writing to HDFS. Key sorting alone accounted for nearly 50% of the runtime in the PageRank benchmark, and serialization another 15%. In contrast, with Piccolo, the need for (1) is eliminated and the overhead associated with (2) and (3) is greatly reduced. PageRank rank values are stored in memory and are available across iterations without being serialized to a distributed file system. In addition, as most outgoing links point to other pages at the same site, a kernel instance ends up performing most updates directly to locally stored table data, thereby avoiding serialization for those updates entirely.

Comparison with MPI: We compared the the performance of matrix multiplication using Piccolo to a third-party MPI-based implementation [2]. The MPI version uses Cannon's algorithm for blocked matrix multiplication and uses MPI specific communication primitives to handle data broadcast and the simultaneous sending and receiving of data. For Piccolo, we implemented the naïve blocked multiplication algorithm, using our distributed tables to handle the communication of matrix state. As Piccolo relies on MPI primitives for communication, we do not expect to see performance advantage, but are more interested in quantifying the amount of overhead incurred.

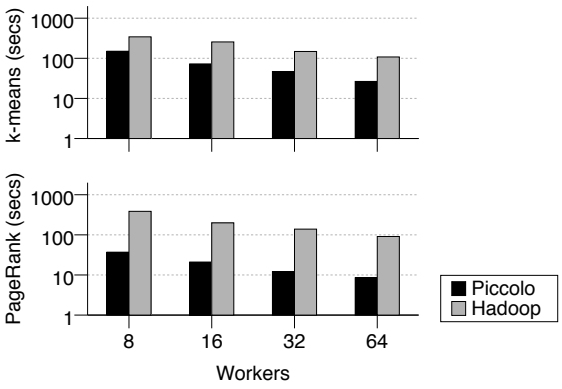


Figure 9: Per-iteration running time of PageRank and *k*-means in Hadoop and Piccolo (fixed default input size).

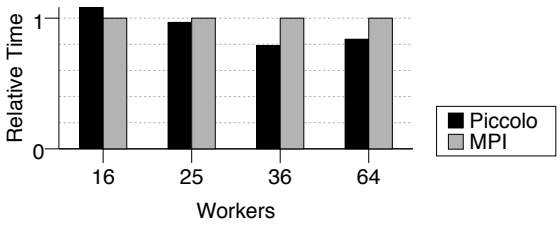


Figure 10: Runtime of matrix multiply, scaled relative to MPI.

Figure 10 shows that the running time of the Piccolo implementation is no more than 10% of the MPI implementation. We were surprised to see that our Piccolo implementation out-performed the MPI version in experiments with more workers. Upon inspection, we found that this was due to slight performance differences between machines in our cluster; as the MPI implementation has many more synchronization points than that of Piccolo, it is forced to wait for slower nodes to catch up.

6.5 Work Stealing and Slow Machines

The PageRank benchmark provides a good basis for testing the effect of work stealing because the web graph partitions have highly variable sizes: the largest partition for the 900M-page graph is 5 times the size of the smallest. Using the same benchmark, we also tested how performance changed when one worker was operating slower then the rest. To do so, we ran a CPU-intensive program on one core that resulted in the worker bound to that core

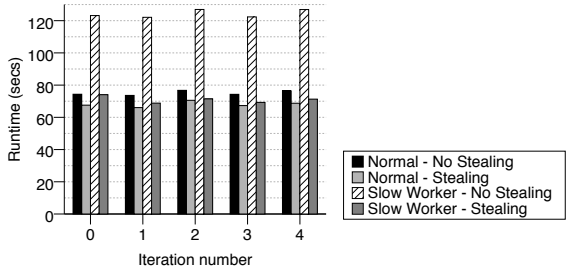


Figure 11: Effect of Work Stealing and Slow Workers

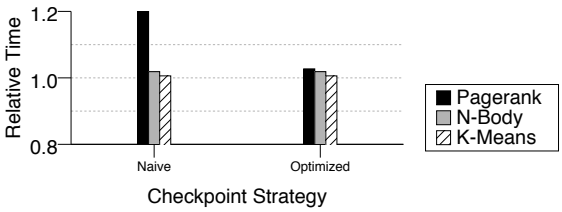


Figure 12: Checkpoint overhead. Per-iteration runtime is scaled relative to without checkpointing.

having only 50% of the CPU time of the other workers.

The results of these tests are shown in Figure 11. Work stealing improves running time by 10% when all machines are operating normally. The improvement is due to the imbalance in the input partition sizes - when run without work stealing, the computation waits longer for the workers processing more data to catch up.

The effect of slow workers on the computation is more dramatic. With work-stealing disabled, the runtime is nearly double that of the normal computation, as each iteration must wait for the slowest worker to complete all assigned tasks. Enabling work stealing improves the situation dramatically - the computation time is reduced to less then 5% over that of the non-slow case.

6.6 Checkpointing

We evaluated the checkpointing overhead using the PageRank, *k*-means and *n*-body problems. Compared to the other problems, PageRank has a larger table that needs to be checkpointed, making it a more demanding test of checkpoint/restore performance. In our experiment, each worker wrote its checkpointed table partitions to the local disk. Figure 12 shows the runtime when checkpointing is enabled relative to when there is no checkpointing. For the naïve synchronous checkpointing strategy, the master starts checkpointing only after all workers have finished. For the optimized strategy, the master initiates the checkpoint as soon as one of the workers has finished. As the figure shows, overhead of the optimized checkpointing strategy is quite negligible ($\sim$2%) and the optimization of starting checkpointing early results in significant reduction of overhead for the larger PageRank checkpoint.

Limitations of global checkpoint and restore: The global nature of Piccolo's failure recovery mechanism raises the question of scalability. As the of a cluster increases, failure becomes more frequent; this causes more frequent checkpointing and restoration which consume a larger fraction of the overall computation time. While we lacked the machine resources to directly test the performance of Piccolo on thousands of machines, we estimate scalability limit of Piccolo's checkpointing mechanism

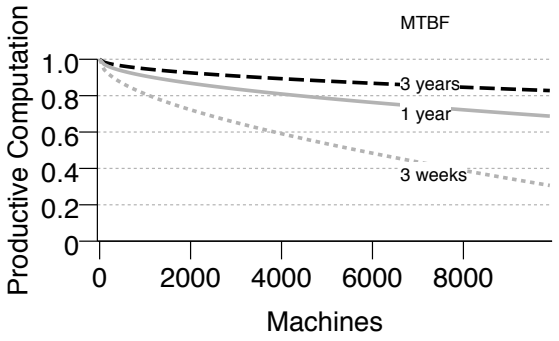


Figure 13: Expected scaling for large clusters.

based on expected machine uptime.

We consider a hypothetical cluster of machines with 16GB of RAM and 4 disk drives. We measured the time taken to checkpoint and restore such a machine in the "worst case" - a computation whose table state uses all available system memory. We estimate the fraction of time a Piccolo computation would spend working productively (not in a checkpoint or restore state), for varying numbers of machines and failure rates. In our model, we assume that machine failures arrive at a constant interval defined by the failure rate and the number of machines in a cluster. While this is a simplification of real-life failure behavior, it is a worst-case scenario for the restore mechanism, and as such provides a useful lower bound. The expected efficiency based on our model is shown in Figure 13. For well maintained data-centers that we are familiar with, the average machine uptime is typically around 1 year. For these data-centers, the global checkpointing mechanism can efficiently scale up to a few thousand machines.

6.7 Distributed Crawler

We evaluated our distributed crawler implementation using various numbers of workers. The *URL* table was initialized with a seed set of 1000 URLs. At the end of a 30 minutes run of the experiment, we measured the number of pages crawled and bytes downloaded. Figure 14 shows the crawler's web page download throughput in

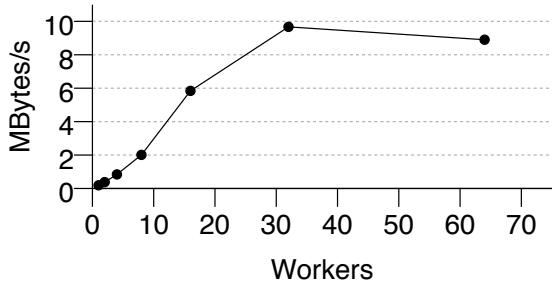


Figure 14: Crawler throughput

MBytes/sec as N increases from 1 to 64. The crawler spends most CPU time in the Python code for parsing HTML and URLs. Therefore, its throughput scales approximately linearly with N. At N=32, the crawler download throughput peaks at ∼10MB/s which is limited by our 100-Mbps Internet uplink. There are highly optimized single-server crawler implementations that can sustain higher download rates than 100Mbps [49]. However, our Piccolo-based crawler could potentially scale to even higher download rates despite being built using Python.

7 Related Work

Communication-oriented models: Communication-based primitives such as MPI [21] and Parallel Virtual Machine (PVM [46]) have been popular for constructing distributed programs for many years. MPI and PVM offer extensive messaging mechanisms including unicast and broadcast as well as support for creating and managing remote processes in a distributed environment. There has been continuous research on developing experimental features for MPI, such as optimization of collective operations [3], fault-tolerance via machine virtualization [34] and the use of hybrid checkpoint and logging for recovery [10]. MPI has been used to build very high performance applications - its support of explicit communication allows considerable flexibility in writing applications to take advantage of a wide variety of network topologies in supercomputing environments. This flexibility has a cost in the form of complexity - users must explicitly manage communication and synchronization of state between workers, which can become difficult to do while attempting to retain efficient and correct execution.

BSP (Bulk Synchronous Parallel) is a high-level communication-oriented model [50]. In this model, threads execute on different processors with local memory, communicate with each other using messages, and perform global-barrier synchronization. BSP implementations are typically realized using MPI [25]. Recently, the BSP model has been adopted in the Pregel framework for parallelizing work on large graphs [33].

Distributed shared-memory: The complexity of programming for communication-oriented models drove a wave of research in the area of distributed shared memory (DSM) systems [30, 29, 32, 7]. Most DSM systems aim to provide *transparent* memory access, which causes programs written for DSMs to incur many fine-grained synchronization events and remote memory reads. While initially promising, DSM research has fallen off as the ratio of network latency to local CPU performance has widened, making naïve remote accesses and synchronization prohibitively expensive.

Parallel Global Address Space (PGAS) [17, 35, 51] are a set of language extensions to realize a distributed shared address space. These extensions try to ameliorate the latency problems of DSM by allowing users to express affinities of portions of shared memory with a particular thread, thereby reducing the frequency of remote memory references. They retain the low level (flat memory) interface common to DSM. As a result, applications written for PGAS systems still require fine-grained synchronization when operating on non-primitive datatypes, or in order to aggregate several values (for instance, computing the sum of a memory location with multiple writers).

Tuple spaces, as seen in coordination languages such as Linda [13] and more recently JavaSpaces [22], expose to users a global tuple-space accessible from all participating threads. Although tuple spaces provide atomic primitives for reading and writing tuples, they are not intended for high-frequency access. As such, there is no support for locality optimization nor write-write conflict resolution.

MapReduce and Dataflow models: In recent years, MapReduce has emerged as a popular programming model for parallel data processing [19]. There are many recent efforts inspired by MapReduce ranging from generalizing MapReduce to support the join operation [27], improving MapReduce's pipelining performance [16], building high-level languages on top of MapReduce (e.g. DryadLINQ [53], Hive [48], Pig [37] and Sawzall [40]). FlumeJava [14] provides a set of collection abstractions and parallel execution primitives which are optimized and compiled down to a sequence of MapReduce operations.

The programming models of MapReduce [19] and Dryad [27] are instances of stream processing, or data-flow models. Because of MapReduce's popularity, programmers start using it to build in-memory iterative applications such as PageRank, even though the data-flow model is not a natural fit for these applications. Spark [54] proposes to add distributed read-only in-memory cache to improve the performance of

MapReduce-based iterative computations.

Single-machine shared memory models: Many programming models are available for parallelizing execution on a single machine. In this setting, there exists a physically-shared memory among computing cores supporting low-latency memory access and fast synchronization between threads of computation, which are not available in a distributed environment. Although there are also popular streaming/data-flow models [44, 47, 12], most parallel models for a single machine are based on shared-memory. For the GPU platform, there are CUDA [36] and OpenCL [24]. For multicore CPUs, Cilk [8] and more recently, Intel's Thread Building Blocks [41] provide support for low-overhead thread creation and dispatching of tasks at a fine level. OpenMP [18] is a popular shared-memory model among the scientific computing community: it allows users to target sections of code for parallel execution and provides synchronization and reduction primitives. Recently, there have been efforts to support OpenMP programs across a cluster of machines [26, 5]. However, based on software distributed shared memory, the resulting implementations suffer from the same limitations of DSMs and PGAS systems.

Distributed data structures: The goal of distributed data structures is to provide a flexible and scalable data storage or caching interface. Examples of these include DDS [23], Memcached [39], the recently proposed RamCloud [38], and many key-value stores based on distributed hash tables [4, 20, 45, 42]. These systems do not seek to provide a computation model, but rather are targeted towards loosely-coupled distributed applications such as web serving.

8 Conclusion

Parallel in-memory application need to access and share intermediate state that reside on different machines. Piccolo provides a programming model that supports the sharing of mutable, distributed in-memory state via a key/value table interface. Piccolo helps applications achieve high performance by optimizing for locality of access to shared state and having the run-time automatically resolve write-write conflicts using application-specified accumulation functions.

Acknowledgments

Yasemin Avcular and Christopher Mitchell ran some of the Hadoop experiments. We thank the many people who have improved this work through discussion and reviews: the members of NeWS group at NYU, Frank Dabek, Rob Fergus, Michael Freedman, Robert Grimm, Wilson Hsieh, Frans Kaashoek, Jinyuan Li, Robert Morris, Sam Roweis, Torsten Suel, Junfeng Yang, Nickolai Zeldovich.

References

[1] Apache hadoop. `http://hadoop.apache.org`.

[2] Example matrix multiplication implementation using mpi. `http://www.cs.umanitoba.ca/~comp4510/examples.html`.

[3] ALMÁSI, G., HEIDELBERGER, P., ARCHER, C. J., MARTORELL, X., ERWAY, C. C., MOREIRA, J. E., STEINMACHER-BUROW, B., AND ZHENG, Y. Optimization of MPI collective communication on BlueGene/L systems. In *Proceedings of the 19th annual international conference on Supercomputing* (New York, NY, USA, 2005), ICS '05, ACM, pp. 253–262.

[4] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. FAWN: a fast array of wimpy nodes. In *SOSP* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 1–14.

[5] BASUMALLIK, A., MIN, S.-J., AND EIGENMANN, R. Programming distributed memory sytems using OpenMP. *Parallel and Distributed Processing Symposium, International 0* (2007), 207.

[6] BEAZLEY, D. M. Automated scientific software scripting with SWIG. *Future Gener. Comput. Syst. 19* (July 2003), 599–609.

[7] BERSHAD, B. N., ZEKAUSKAS, M. J., AND SAWDON, W. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE Computer Society International Conference* (1993).

[8] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 1995), ACM, pp. 207–216.

[9] BOLDI, P., AND VIGNA, S. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)* (Manhattan, USA, 2004), ACM Press, pp. 595–601.

[10] BOSILCA, G., BOUTEILLER, A., CAPPELLO, F., DJILALI, S., FEDAK, G., GERMAIN, C., HERAULT, T., LEMARINIER, P., LODYGENSKY, O., MAGNIETTE, F., NERI, V., AND SELIKHOV, A. Mpich-v: toward a scalable fault tolerant mpi for volatile nodes. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (Los Alamitos, CA, USA, 2002), Supercomputing '02, IEEE Computer Society Press, pp. 1–18.

[11] BRIN, S., AND PAGE, L. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems 30*, 1-7 (1998), 107 – 117. Proceedings of the Seventh International World Wide Web Conference.

[12] BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers* (2004), ACM, p. 786.

[13] CARRIERO, N., AND GELERNTER, D. Linda in context. *Commun. ACM 32*, 4 (1989), 444–458.

[14] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R. R., BRADSHAW, R., AND WEIZENBAUM, N. Flumejava: Easy, efficient data-parallel pipelines. In *PLDI - ACM SIGPLAN 2010* (2010).

[15] CHANDY, K. M., AND LAMPORT, L. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS) 3* (1985), 63–75.

[16] CONDIE, T., CONWAY, N., ALVARO, P., AND HELLERSTEIN, J. MapReduce online. In *NSDI* (2010).

[17] CONSORTIUM, U. UPC language specifications, v1.2. Tech. rep., Lawrence Berkeley National Lab, 2005.

[18] DAGUM, L., AND MENON, R. Open MP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering 5*, 1 (1998), 46–55.

[19] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation (OSDI)* (2004).

[20] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles* (Oct. 2007), pp. 205–220.

[21] FORUM, M. MPI 2.0 standard, 1997.

[22] FREEMAN, E., ARNOLD, K., AND HUPFER, S. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[23] GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. Scalable, distributed data structures for internet service construction. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation* (Berkeley, CA, USA, 2000), USENIX Association, pp. 22–22.

[24] GROUP, K. O. W. The OpenCL specification. Tech. rep., 2009.

[25] HILL, J., MCCOLL, W., STEFANESCU, D., GOUDREAU, M., LANG, K., RAO, S., SUEL, T., TSANTILAS, T., AND BISSELING, H. Bsplib: The bsp programming library. *Parallel Computing 24* (1998).

[26] HOEFLINGER, J. P. Extending OpenMP to clusters. Tech. rep., Intel, 2009.

[27] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)* (2007).

[28] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: Fair scheduling for distributed computing clusters. In *SOSP* (2010).

[29] JOHNSON, K. L., KAASHOEK, M. F., AND WALLACH, D. A. CRL: High-performance all-software distributed shared memory. In *SOSP* (1995).

[30] KELEHER, P., COX, A. L., AND ZWAENEPOEL, W. Lazy release consistency for software distributed shared memory. In *In Proceedings of the 19th Annual International Symposium on Computer Architecture* (1992).

[31] LAMPORT, L. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers 28*, 9 (1979).

[32] LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS) 7* (1989), 321–359.

[33] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), ACM, pp. 135–146.

[34] NAGARAJAN, A. B., MUELLER, F., ENGELMANN, C., AND SCOTT, S. L. Proactive fault tolerance for hpc with xen virtualization. In *Proceedings of the 21st annual international conference on Supercomputing* (New York, NY, USA, 2007), ICS '07, ACM, pp. 23–32.

[35] NUMRICH, R. W., AND REID, J. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum 17* (August 1998), 1–31.

[36] NVIDIA. CUDA programming guide (ver 3.0).

[37] OLSON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A not-so-foreign language for data processing. In *ACM SIGMOD* (2008).

[38] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIERES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., RUMBERL, S., STRATMANN, E., AND STUTSMAN, R. The case for RAMclouds: Scalable high-performance storage entirely in DRAM. In *Operating system review* (Dec. 2009).

[39] PHILLIPS, L., AND FITZPATRICK, B. Livejournal's backend and memcached: Past, present, and future. In *LISA* (2004), USENIX.

[40] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. In *Scientific Programming* (2005).

[41] REINDERS, J. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[42] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *18th IFIP/ACM International Conference on Distributed Systems Platforms* (Nov. 2001).

[43] SINGH, J. P., WEBER, W.-D., AND GUPTA, A. SPLASH: Stanford parallel applications for shared-memory. Tech. rep., Stanford University, 1991.

[44] STEPHENS, R. A survey of stream processing, 1995.

[45] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking* (2002), 149–160.

[46] SUNDERAM, V. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience* (1990), 315–339.

[47] THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. StreamIt: A language for streaming applications. In *Compiler Construction* (2002), Springer, pp. 49–84.

[48] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow. 2* (August 2009), 1626–1629.

[49] TSANG LEE, H., LEONARD, D., WANG, X., AND LOGUINOV, D. Irlbot: Scaling to 6 billion pages and beyond. In *WWW Conference* (2008).

[50] VALIANT, L. A bridging model for parallel computation. *Communications of the ACM 33* (1990).

[51] YELICK, K., SEMENZATO, L., PIKE, G., MIYAMOTO, C., LIBLIT, B., KRISHNAMURTHY, A., GRAHAM, P. H. S., GAY, D., COLELLA, P., AND AIKEN, A. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience 10*, 11 (1998).

[52] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *ACM Symposium on Operating Systems Principles (SOSP)* (2009).

[53] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Symposium on Operating System Design and Implementation (OSDI)* (2008).

[54] ZAHARIA, M., CHOWDHURY, N. M. M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Spark: Cluster Computing with Working Sets. Tech. Rep. UCB/EECS-2010-53, EECS Department, University of California, Berkeley, May 2010.

Depot: Cloud storage with minimal trust

Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish
The University of Texas at Austin, fuss@cs.utexas.edu

Abstract

The paper describes the design, implementation, and evaluation of Depot, a cloud storage system that minimizes trust assumptions. Depot tolerates buggy or malicious behavior by *any number* of clients or servers, yet it provides safety and liveness guarantees to correct clients. Depot provides these guarantees using a two-layer architecture. First, Depot ensures that the updates observed by correct nodes are consistently ordered under Fork-Join-Causal consistency (FJC). FJC is a slight weakening of causal consistency that can be both safe and live despite faulty nodes. Second, Depot implements protocols that use this consistent ordering of updates to provide other desirable consistency, staleness, durability, and recovery properties. Our evaluation suggests that the costs of these guarantees are modest and that Depot can tolerate faults and maintain good availability, latency, overhead, and staleness even when significant faults occur.

1 Introduction

This paper describes the design, implementation, and evaluation of Depot, a cloud storage system in the spirit of S3 [1], Azure [4], and Google Storage [3] but with a crucial difference: Depot clients do not have to *trust*, that is *assume*, that Depot servers operate correctly.

What motivates Depot is that cloud storage service providers (SSPs), such as S3 and Azure, are fault-prone black boxes operated by a party other than the data owner. Indeed, clouds can experience software bugs [9], correlated manufacturing defects [57], misconfigured servers and operator error [53], malicious insiders [68], bankruptcy [5], undiagnosed problems [14], Acts of God (e.g., fires [20]) and Man [50]. Thus, it seems prudent for clients to avoid strong assumptions about an SSP's design, implementation, operation, and status—and instead to rely on end-to-end checks of well-defined properties. In fact, removing such assumptions promises to help SSPs too: today, a significant barrier to adopting cloud services is precisely that many organizations hesitate to place trust in the cloud [18].

Given this motivation, Depot assumes less than any prior system about the correctness of participating hosts:

- *Depot **eliminates** trust for safety.* A client can ensure safety by assuming the correctness of only itself. Depot guarantees that any subset of correct clients observes sensible, well-defined semantics. This holds regardless of how many nodes fail and no matter whether they are clients or servers, whether these are failures of omission or commission, and whether these failures are accidental or malicious.
- *Depot **minimizes** trust for liveness and availability.* We wish we could say "trust only yourself" for liveness and availability. Depot does eliminate trust for updates: a client can always update any object for which it is authorized, and any subset of connected, correct clients can always share updates. However, for reads, there is a fundamental limit to what any storage system can guarantee: if no correct, reachable node has an object, that object may be unavailable. We cope with this fundamental limit by allowing reads to be served by any node (even other clients) while preserving the system's guarantees, and by configuring the replication policy to use several servers (which protects against failures of clients and subsets of servers) and at least one client (which protects against temporary [8] and permanent [5, 14] cloud failures).

Though prior work has reduced trust assumptions in storage systems, it has not minimized trust with respect to safety, liveness, or both. For example, quorum and replicated state machine approaches [15, 19, 30] tolerate failures by a fraction of servers. However, they sacrifice safety when faults exceed a threshold and liveness when too few servers are reachable. Fork-based systems [12, 13, 43, 44] remain safe without trusting a server, but they compromise liveness in two ways. First, if the server is unreachable, clients must block. Second, a faulty server can permanently partition correct clients, preventing them from ever observing each other's subsequent updates.

Indeed, it is challenging to guarantee safety and liveness while minimizing trust assumptions: without some assumptions about correct operation, providing even a weak guarantee like eventual consistency—the bare minimum of what a storage service should provide—seems difficult. For example, a faulty storage node receiving an update from a correct client might quietly fail to propagate that update, thereby hiding it from the rest of the system. Perhaps surprisingly, we find that eventual consistency *is* possible in this environment.

In fact, Depot meets a contract far stronger than eventual consistency even under assorted and abundant faults and failures. This set of well-defined guarantees under weak assumptions is Depot's top-level contribution, and it derives from a novel synthesis of prior mechanisms and our own. Depot is built around three key ideas:

(1) Reduce misbehavior to concurrency. As in prior work [12, 13, 43, 44], the protocol requires that an update be signed and that it name both its antecedents and the system state seen by the updater. Then, misbehavior by clients or servers is limited to *forking*: showing divergent histories to different nodes. However, previous work *detects* but does not *repair* forks. In contrast, Depot allows correct clients to *join forks*, that is, to incorporate the divergence into a sensible history, which allows them to keep operating in the face of faults. Specifically, a correct node regards a fork as logically concurrent updates by two *virtual nodes*. At that point, correct nodes can handle forking by faulty nodes using the same techniques [11, 23, 37, 61, 67] that they need anyway to handle a better understood problem: logically concurrent updates during disconnected operation.

(2) Enforce Fork-Join-Causal consistency. To allow end-to-end checks on SSP behavior, we must specify a contract: When must an update be visible to a read? When is it okay for a read to "miss" a recent update? Depot guarantees that a correct client observes *Fork-Join-Causal consistency* (FJC) no matter how many other nodes are faulty. FJC is a slight weakening of causal consistency [7, 40, 56]. Depot defines FJC as its consistency contract because it is weak enough to enforce despite faulty nodes and without hurting availability. At the same time, FJC is strong enough to be useful: nodes see each other's updates in an order that reflects dependencies among both correct and faulty nodes' writes. This ordering is useful not only for end users of Depot but also internally, within Depot.

(3) Layer other storage properties over FJC. Depot implements a layered architecture that builds on the ordering guarantees provided by FJC to provide other desirable properties: eventual consistency, bounded staleness, durability, high availability, integrity (ensuring that only authorized nodes can update an object), snapshotting of versions (to guard against spurious updates from faulty clients), garbage collection, and eviction of faulty nodes.[1] For all of these properties, the challenge is to precisely define the strongest guarantee that Depot can provide with minimal assumptions about correct operation. Once each property is defined, implementation is straightforward because we can build on FJC, which lets us reason about the order in which updates propagate through the system.

The price of providing these guarantees is tolerable, as demonstrated by an experimental evaluation of a prototype implementation of Depot. Depot adds a few hundred bytes of metadata to each update and each stored object, and it requires a client to sign and store each of its updates. We demonstrate that Depot can tolerate faults and maintain good availability, latency, overhead, and staleness even when significant faults occur. Additionally, because Depot makes minimal assumptions about servers, we can implement *Teapot*, a variation of Depot that provides many of Depot's guarantees using an unmodified SSP, such as Amazon's S3. The difference between Depot and Teapot suggests several modest extensions to SSPs' interfaces that would strengthen their guarantees.

[1] We are not explicitly addressing confidentiality and privacy, but, as discussed in §3.1, existing approaches can be layered on Depot.

2 Why untrusted storage?

When we say that a component is untrusted, we are not adopting a "tinfoil hat" stance that the component is operated by a malicious actor, nor are we challenging the honesty of storage service providers. What we mean is that the system provides guarantees, usually achieved by end-to-end checks, even if the given component is incorrect. Since components could be incorrect for many reasons (as stated in the introduction), we believe that designing to tolerate incorrectness is prudence, not paranoia. We now answer some natural questions.

SSPs are operated by large, reputable companies, so why not trust them? That is like asking, "Banks are large, reputable repositories of money, so why do we need bank statements?" For many reasons, customers *and* banks want customers to be able to check the bank's view of their account activity. Likewise, our approach might appeal not only to customers but also to SSPs: by requiring less trust, a service might attract more business.

How likely are faults in the SSP? We do not know the precise probability. However, we know that providers do fail (as mentioned in the introduction). More broadly, they carry non-negligible risks. First, they are opaque (by nature). Second, they are complex distributed systems. Indeed, coping with known hardware failure modes in *local* file systems is difficult [59]; in cloud storage, this difficulty can only grow. Given the opacity and complexity, it seems prudent not to assume the unfailing correctness of an SSP's internals.

Even if we do not assume that SSPs are perfect, the most likely failure is the occasional corrupted or lost block, which can be addressed with checksums and replication. Do you really need mechanisms to handle other cases (that all of the nodes are faulty, that a fork happens, that old or out-of-order data is returned, etc.)? Replication and checksums are helpful, and they are part of Depot. However, they are not sufficient. First, failures are often correlated: as Vogels notes, uncorrelated failures are "absolutely unrealistic ... as [failures] are often triggered by external or environmental events" [69]. These events include the litany in the introduction.

Second, other types of failures are possible. For example, a machine that loses power after failing to commit its output [52, 72] may lose recent updates, leading to forks in history. Or, a network failure might delay propagation

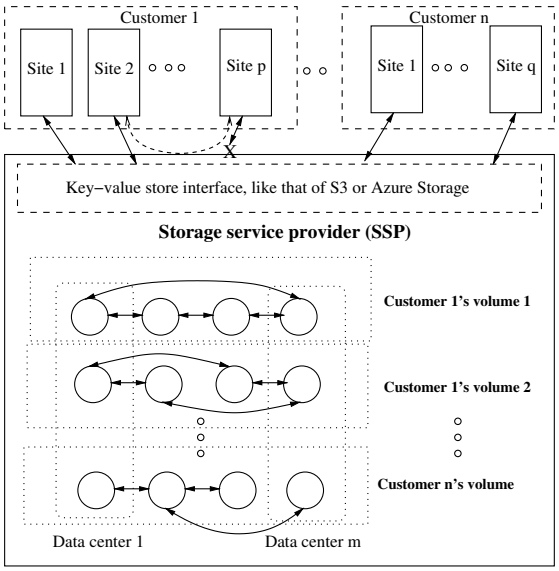


FIG. 1—Architecture of Depot. The arrows between servers indicate replication and exchange.

of an update from one SSP node to another, causing some clients to read stale data. In general, our position is that rather than try to handle every possible failure individually, it is preferable to define an end-to-end contract and then design a system that always meets that contract.

The above events seem unlikely. Is tolerating them worth the cost? One of our purposes in this paper is to report for the first time what that cost is. Whether to "purchase" the guarantees is up to the application, but as the price is modest, we anticipate, with hope, that many applications will find it attractive.

What about clients? We also minimize trust of clients (since they are, of course, also vulnerable to faults).

3 Architecture, scope, and use

Figure 1 depicts Depot's high-level architecture. A set of clients stores key-value pairs on a set of servers. In our target scenario, the servers are operated by a storage service provider (SSP) that is distinct from the data owner that operates the clients.[2] Keys and values are arbitrary strings, with overhead engineered to be low when values are at least a few KB. A Depot client exposes an interface of GET and PUT to its application users.

For scalability, we slice the system into groups of servers, with each group responsible for one or more *volumes*. Each volume corresponds to a range of one customer's keys, and a server independently runs the protocol for each volume assigned to it. Many strategies for partitioning keys are possible [22, 36, 51], and we leave the assignment of keys to volumes to layers above Depot.

The servers for each volume may be geographically distributed, a client can access any server, and servers replicate updates using any topology (chain, mesh, star, etc.). As in Dynamo [22], to maximize availability, Depot does not require overlapping read and write quorums. In fact, as the dotted lines suggest, Depot can even function under complete server unavailability: the protocol permits clients to communicate directly with each other. If the SSP later recovers, clients can continue using the SSP (after sending the missed updates to the servers). This raises a question: why have the SSP at all? We point to the usual benefits of cloud services: cost, scalability, geographic replication, and management.

We use the term *node* to mean either a client or a server. Clients and servers run the same basic Depot protocol, though they are configured differently.

3.1 Issues addressed

One of our aims in this work is to push the envelope in the trade-offs between trust assumptions and system guarantees. Specifically, for a set of standard properties that one might desire in a storage system, we ask: what is the minimum assumption that we need to provide useful guarantees, and what are those guarantees? The issues that we examine are as follows:

- *Consistency* (§4–§5.2) and *bounded staleness* (§5.4): Once a write occurs, the update should be visible to reads "soon". Consistency and staleness properties limit the extent to which the storage system can reorder, delay, or omit making updates visible to reads.
- *Availability and durability* (§5.3): Our availability goal is to maximize the fraction of time that a client succeeds in reading or writing an object. Durability means that the system does not permanently lose data.
- *Integrity and authorization* (§5.5): Only clients authorized to update an object should be able to create valid updates that affect reads on that object.
- *Data recovery* (§5.6): Data owners care about end-to-end reliability. Consistency, durability, and integrity are not enough when the layers above Depot—faulty clients, applications, or users—can issue authorized writes that replace good data with bad. Depot does not try to distinguish good updates from bad ones, nor does it innovate on the abstractions used to defend data from higher-layer failures. We do however explore how Depot can support standard techniques such as *snapshots* to recover earlier versions of data.
- *Evicting faulty nodes* (§5.7): If a faulty node provably deviates from the protocol, we wish to evict it from the system so that it will not continue to disrupt operation. However, we must never evict correct nodes.

[2] Because Depot does not require nodes to trust each other, different data centers in Figure 1 could be operated by different SSPs. Doing so might reduce the risk of correlated failures across replicas [6, 38]. For simplicity, we describe and evaluate only single-SSP configurations.

Depot provides the above properties with a layered approach. Its core protocol (§4) addresses *consistency*. Specifically, the protocol enforces Fork-Join-Causal consistency (FJC), which is the same as causal consistency [7, 40, 56] in benign runs. This protocol is the essential building block for the other properties listed above. In §5, we define these properties precisely and discuss how Depot provides them.

Note that we explicitly do not try to solve the confidentiality/privacy problem within Depot. Instead, like commercial storage systems [1, 4], Depot enforces integrity and authorization (via client signatures) but leaves it to higher layers to use appropriate techniques for the privacy requirements of each application (e.g., allow global access, encrypt values, encrypt both keys and values, introduce artificial requests to thwart traffic analysis, etc.).

We also do not claim that the above list of issues is exhaustive. For example, it may be useful to audit storage service providers with black box tests to verify that they are storing data as promised [38, 62], but we do not examine that issue. Still, we believe that the properties are sufficient to make the resulting system useful.

3.2 Depot in use: Applications & conflicts

Depot's key-value store is a low-level building block over which many applications can be built. For example, hundreds of widely used applications—including backup, point of sale software, file transfer, investment analytics, cross-company collaboration, and telemedicine—use the S3 key-value store [2], and Depot can serve all of them: it provides a similar interface to S3, and it provides strictly stronger guarantees.

An issue in systems that are causally consistent and weaker—a set that includes not just Depot and S3 but also CVS, SVN, Git, Bayou [56], Coda [37], and others—is handling concurrent writes to the same object. Such *conflicts* are unfortunate but unavoidable: they are provably the price of high availability [26].

Many approaches to resolving conflicting updates have been proposed [37, 61, 67], and Depot does not claim to extend the state of the art on this front. In fact, Depot is less ambitious than some past efforts: rather than try to resolve conflicts internally (e.g., by picking a winner, merging concurrent updates, or rolling back and re-executing transactions [67]), Depot simply exposes concurrency when it occurs: a read of key k returns the *set* of updates to k that have not been superseded by any logically later update of k.[3]

This approach is similar to that of S3's replication substrate, Dynamo [22], and it supports a range of application-level policies. For example, applications using Depot may resolve conflicts by *filtering* (e.g., reads return the update by the highest-numbered node, reads return an application-specific merge of all updates, or reads return all updates) or by *replacing* (e.g., the application reads the multiple concurrent values, performs some computation on them, and then writes a new value that thus appears logically after, and thereby supersedes, the conflicting writes).

[3]Note that Depot neither creates concurrency nor makes the problem worse. If an application cannot deal with conflicts, it can still use Depot but must restrict its use (e.g., by adding locks and sending all operations through a single SSP node), and it must sacrifice the ability to tolerate faults (such as forks) that appear as concurrency.

3.3 System and threat model

We now briefly state our technical assumptions. First, nodes are subject to standard cryptographic hardness assumptions, and each node has a public key known to all nodes. Second, *any number* of nodes can fail in arbitrary (Byzantine [41]) ways: they can crash, corrupt data, lose data, process some updates but not others, process messages incorrectly, collude, etc. Third, we assume that any pair of timely, connected, and correct nodes can eventually exchange any finite number of messages. That is, a faulty node cannot forever prevent two correct nodes from communicating (but we make no assumptions about how long "eventually" is).

Fourth, above we used the term *correct node*. This term refers to a node that never deviates from the protocol nor becomes permanently unavailable. A node that obeys the protocol for a time but later deviates is not counted as correct. Conversely, a node that crashes and recovers with committed state intact is equivalent to a correct node that is slow. Fifth, to ensure the liveness of garbage collection, we assume that unresponsive clients are eventually repaired or replaced. To satisfy this assumption, an administrator can install an unresponsive client's keys and configuration on new hardware [15].

4 Core protocol

In Depot, clients' reads and updates to shared objects should always appear in an order that reflects the logic of higher layers. For example, an update that removes one's parents from a friend list and an update that posts spring break photos should appear in that order, not the other way around [21]. However, Depot has two challenges. First, it aims for maximum availability, which fundamentally conflicts with the strictest orderings [26]. Second, it aims to provide its ordering guarantees despite arbitrary misbehavior from any subset of nodes. In this section, we describe how the protocol at Depot's core achieves a sensible and robust order of updates while optimizing for availability and tolerating arbitrary misbehavior.

As mentioned above, this basic protocol is run by both clients and servers. This symmetry not only simplifies the design but also provides flexibility. For example, if

servers are unreachable, clients can share data directly. For simplicity, the description below does not distinguish between clients and servers.

4.1 Basic protocol

This subsection describes the basic protocol to propagate updates, ignoring the problems raised by faulty nodes. The protocol is essentially a standard log exchange protocol [10, 56]; we describe it here for background and to define terms.

The core message in Depot is an *update* that changes the *value* associated with a *key*. It has the following form:

$$\text{dVV, } \{\text{key, H(value), logicalClock@nodeID, H(history)}\}_{\sigma_{\text{nodeID}}}$$

Updates are associated with logical times. A node assigns each update an *accept stamp* of the form *logicalClock@nodeID* [56]. A node N increments its logical clock on each local write. Also, when N receives an update u from another node, N advances its logical clock to exceed u's. Thus, an update's accept stamp exceeds the accept stamp of any update on which it depends [40]. The remaining fields, *dVV* and *H(history)*, and the writer's signature, σ_{nodeID}, defend against faults and are discussed in subsections 4.2 and 4.3.

Each node maintains two local data structures: a *log* of updates it has seen and a *checkpoint* reflecting the current state of the system. For efficiency, Depot separates data from metadata [10], so the log and checkpoint contain collision-resistant hashes of values. If a node knows the hash of a value, it can fetch the full value from another node and store the full value in its checkpoint. Each node sorts the updates in its log by accept stamp, sorting first by *logicalClock* and breaking ties with *nodeID*. Thus, each new write issued by a node appears at the end of its own log and (assuming no faulty nodes) the log reflects a causally consistent ordering of all writes.

Information about updates propagates through the system when nodes exchange tails of their logs. Each node N maintains a *version vector VV* with an entry for each node M in the system: $N.VV[M]$ is the highest logical clock N has observed for any update by M [55]. To transmit updates from node M to node N, M sends to N the updates from its log that N has not seen.

Two updates are *logically concurrent* if neither appears in the other's history. Concurrent writes may *conflict* if they update the same object; conflicts are handled as described in Section 3.2.

4.2 Consistency despite faults

There are three fields in an update that defend the protocol against faulty nodes. The first is a *history hash*, *H(history)*, that encodes the history on which the update depends using a collision-resistant hash that covers the most recent update by each node known to the writer when it issued the update. By recursion, this hash covers all updates included by the writer's current version vector. Second, each update is sent with a dependency version vector, *dVV*, that indicates the version vector that the history hash covers. Note that while *dVV* logically represents a full version vector, when node N creates an update u, u's *dVV* actually contains only the entries that have changed since the last write by N. Third, a node signs its updates with its private key.

A correct node C accepts an update u only if it meets five conditions. First, u must be properly signed. Second, except as described in the next subsection, u must be newer than any updates from the signing node that C has already received. This check prevents C from accepting updates that modify the history of another node's writes. Third, C's version vector must include u's *dVV*. Fourth, u's *history hash* must match a hash computed by C across every node's last update at time *dVV*. The third and fourth checks ensure that before receiving update u, C has received all of the updates on which u depends. Fifth, u's accept stamp must be at most a constant times C's current wall-clock time (e.g., $u.acceptStamp < 1000 * currentTimeMillis()$). This check defends against exhaustion of the 64-bit logical time space.

Given these checks, attempts by a faulty node to fabricate u and pass it as coming from a correct node, to omit updates on which u depends, or to reorder updates on which u depends will result in C rejecting u. To compromise causal consistency, a faulty node has one remaining option: to *fork*, that is, to show different sequences of updates to different communication partners [43]. Such behavior certainly damages consistency. However, the mechanisms above limit that damage, as we now illustrate with an example. Then, in subsection 4.3 we describe how Depot *recovers* from forks.

Example: The history hash in action A faulty node M can create two updates $u_{1@M}$ and $u'_{1@M}$ such that neither update's history includes the other's. M can then send $u_{1@M}$ and the updates on which it depends to one node, $N1$, and $u'_{1@M}$ and its preceding updates to another node, $N2$. $N1$ can then issue new updates that depend on updates from *one* of M's forked updates (here, $u_{1@M}$) and send these new updates to $N2$. At this point, absent the history hash, $N2$ would receive $N1$'s new updates without receiving the updates by M on which they depend: $N2$ already received $u'_{1@M}$, so its version vector appears to already include the prior updates. Then, if $N2$ applies just $N1$'s writes to its log and checkpoint, multiple consistency violations could occur. First, the system may never achieve eventual consistency because $N2$ may never see write $u_{1@M}$. Further, the system may violate causality because $N2$ has updates from $N1$ but not some earlier updates (e.g., $u_{1@M}$) on which they depend.

The above confusion is prevented by the history hash.

If $N1$ tries to send its new updates to $N2$, $N2$ will be unable to match the new updates' history hashes to the updates $N2$ actually observed, and $N2$ will reject $N1$'s updates (and vice-versa). As a result, $N1$ and $N2$ will be unable to exchange any updates after the *fork junction* introduced by M after $u_{0@M}$.

Discussion At this point, we have composed mechanisms from Bayou [56] and PRACTI [10] (update exchange), SUNDR [43] (signed version vectors), and BFT2F [44] (history hashes, here used by *clients* and modified to apply to history trees instead of linear histories) to provide *fork-causal consistency* (FCC) under arbitrary faults. We define FCC precisely in a technical report [45]. Informally, it means that each node sees a causally consistent subset of the system's updates even though the whole system may no longer be causally consistent. Thus, although the global history has branched, as each node peers backward from its branch to the beginning of time, it sees causal events the entire way.

Unfortunately, enforcing even this weakening of causal consistency would prohibit eventual consistency, crippling the system: FCC requires that once two nodes have been forked, they can never observe one another's updates after the fork junction [43]. In many environments, partitioning nodes this way is unacceptable. In those cases, it would be far preferable to further weaken consistency to ensure an availability property: *connected, correct nodes can always share updates*. We now describe how Depot achieves this property, using a new mechanism: *joining forks* in the system's history.

4.3 Protecting availability: Joining forks

To *join forks*, nodes use a simple coping strategy: they convert concurrent updates by a single faulty node into concurrent updates by a pair of virtual nodes. A node that receives these updates handles them as it would "normal" concurrency: it applies both sets of updates to its state and, if both branches modify the same object, it returns both conflicting updates on reads (§3.2). We now fill in some details.

Identifying a fork First consider a two-way fork. A fork junction comprises exactly three updates where a faulty node M has created two updates (e.g., $u_{1@M}$ and $u'_{1@M}$) such that (i) neither update includes the other in its history and (ii) each update's history hash links it to the same previous update by that writer (e.g., $u_{0@M}$). If a node $N2$ receives from a node $N1$ an update whose history is incompatible with the updates it has already received, and if neither node has yet identified the fork junction, $N1$ and $N2$ identify the three forking updates as follows. First, $N1$ and $N2$ perform a binary search on the updates included in the nodes' version vectors to identify the latest version vector, VV_c, encompassing a common history. Then, $N1$ sends its log of updates beginning from VV_c. Finally, at some point, $N2$ receives the first update by M (e.g., $u_{1@M}$) that is incompatible with the updates by M that $N2$ has already received (e.g., $u_{0@M}$ and $u'_{1@M}$).

Tracking forked histories After a node identifies the three updates in the fork junction, it expands its version vector to include three entries for the node that issued the forking updates. The first is the pre-fork entry, whose index is the index (e.g., M) before the fork and whose contents will not advance past the logical clock of the last update before the fork (e.g., $u_{0@M}$). The other two are the post-fork entries, whose indices consist of the index before the fork augmented with the history hash of the respective first update after the fork. Each of these entries initially holds the logical clock of the first update after the fork (e.g., of $u_{1@M}$ and $u'_{1@M}$); these values advance as the node receives new updates after the fork junction.

Note that this approach works without modification if a faulty node creates a j-way fork, creating updates $u^1_{1@M}$, $u^2_{1@M}$, ..., $u^j_{1@M}$ that link to the same prior update (e.g., $u_{0@M}$). The reason is that, regardless of the order in which nodes detect fork junctions, the branches receive identical names (because branches are named by the first update in the branch). A faulty node that is responsible for multiple dependent forks does not stymie this construction either. After i dependent forks, a virtual node's index in the version vector is well-defined: it is $M \parallel H(u_{fork_1}) \parallel H(u_{fork_2}) \parallel \ldots \parallel H(u_{fork_i})$ [56].

Log exchange revisited The expanded version vector allows a node to identify which updates to send to a peer. In the standard protocol, when a node $N2$ wants to receive updates from $N1$, it sends its current version vector to $N1$ to identify which updates it needs. After $N2$ detects a fork and splits one version vector entry into three, it simply includes all three entries when asking $N1$ for updates. Note that $N1$ may not be aware of the fork, but the history hashes that are part of the indices of $N2$'s expanded version vector (as per the virtual node construction above) tell $N2$ to which branch $N1$'s updates should be applied and tell $N1$ which updates to actually send. Conversely, if the sender $N1$ has received updates that belong to neither branch, then $N1$ and $N2$ identify the new fork junction as described above.

Bounding forks The overhead of this coping strategy is the space, bandwidth, and computation needed for fork detection and larger version vectors. Depot bounds the number of forks that faulty nodes can introduce by (1) making nodes "vouch" for updates by a forking node that they had received before learning of the fork and (2) making them promise not to communicate with known forking nodes. We omit the details for space.

Dimension	Safety/ Liveness	Property	Correct nodes required
Consistency	Safety	Fork-Join Causal	Any subset
	Safety	Bounded staleness	Any subset
	Safety	Eventual consistency (s)	Any subset
Availability	Liveness	Eventual consistency (l)	Any subset
	Liveness	Always write	Any subset
	Liveness	Always exchange	Any subset
	Liveness	Write propagation	Any subset
	Liveness	Read availability / durability	A correct node has object
Integrity	Safety	Only auth. updates	Clients
Recoverability	Safety	Valid discard	1 correct client
Eviction	Safety	Valid eviction	Any subset

FIG. 2—Summary of properties provided by Depot.

5 Properties and guarantees

This section describes how Depot enforces needed properties with minimal trust assumptions. Figure 2 summarizes these properties and lists the required assumptions. Below, we define these properties and describe how Depot provides them. The key idea is that the replication protocol enforces *Fork-Join-Causal consistency* (FJC). Given FJC, we can constrain and reason about the order that updates propagate and use those constraints to help enforce the remaining properties.

5.1 Fork-Join-Causal consistency

Clients expect a storage service to provide consistent access to stored data. Depot guarantees a new consistency semantic for all reads and updates to a volume that are observed by any correct node: *Fork-Join-Causal consistency* (FJC). A formal description of FJC appears in our technical report [45]. Here we describe its core property:

- *Dependency preservation.* If update u_1 by a correct node depends on an update u_0 by any node, then u_0 becomes *observable* before u_1 at any correct node. (An update u of an object o is *observable* at a node if a read of o would return a version at least as new as u [25].)

To explain FJC, we contrast it with causal consistency (CC) in fail-stop systems [7, 40, 56]. CC is based on a dependency preservation property that is identical to the one above, except that it omits the "correct nodes" qualification. Thus, to applications and users, FJC appears almost identical to causal consistency with two exceptions. First, under FJC, a faulty node can issue *forking writes* w and w' such that one correct node observes w without first observing w' while another observes w' without first observing w. Second, under FJC, faulty nodes can issue updates whose stated histories do not include all updates on which they actually depend. For example, when creating the forking updates w and w' just described, the faulty node might have first read updates u_{C1} and u_{C2} from nodes $C1$ and $C2$, then created w that claimed to depend on u_{C1} but not u_{C2}, and finally created update w' that claimed to depend on u_{C2} but not u_{C1}. Note, however, that once a correct node observes w (or w'), it will include w (or w') in its subsequent writes' histories. Thus, as correct nodes observe each others' writes, they will also observe both w and w' and their respective dependencies in a consistent way. Specifically, w and w' will appear as causally concurrent writes by two virtual nodes (§4.3).

Though FJC is weaker than linearizability, sequential consistency, or causal consistency, it still provides properties that are critical to programmers. First, FJC implies a number of useful *session guarantees* [66] for programs at correct nodes, including monotonic reads, monotonic writes, read-your-writes, and writes-follow-reads. Second, as we describe in the subsections below, FJC is the foundation for eventual consistency, for bounded staleness, and for further properties beyond consistency.

Stronger consistency during benign runs. Depot guarantees FJC even if an arbitrary number of nodes fail in arbitrary ways. However, it provides a stronger guarantee—causal consistency—during runs with only omission failures. Of course, causal consistency itself is weaker than sequential consistency or linearizability. We accept this weakening because it allows Depot to remain available to reads and writes during partitions [22, 26].

5.2 Eventual consistency

The term *eventual consistency* is often used informally, and, as the name suggests, it is usually associated with both liveness ("eventual") and safety ("consistency"). For precision, we define eventual consistency as follows.

- *Eventual consistency (safety).* Successful reads of an object at correct nodes that observe the same set of updates return the same values.
- *Eventual consistency (liveness).* Any update issued or observed by a correct node is eventually observable by all correct nodes.

The safety property is directly implied by FJC. The liveness property is ensured by the replication protocol (§4), which entangles updates to prevent selective transmission, and by the communication heuristics (§6), which allow a node that is unable to communicate with a server to communicate with any other server or client.

5.3 Availability and durability

In this subsection, we consider availability of reads, of writes, and of update propagation. We also consider durability. We begin by noting that the following strong availability properties follow from the protocol in §4 and the communication heuristics (§6):

- *Always write.* An authorized node can always update any object.

- *Always exchange.* Any subset of correct nodes can exchange any updates that they have observed, assuming they can communicate as per our model in §3.3.
- *Write propagation.* If a correct node issues a write, eventually all correct nodes observe that write, assuming that any message sent between correct nodes is eventually delivered.

Unfortunately, there is a limit to what any storage system can guarantee for *reads*: if no correct node has an object, then the object may not be durable, and if no correct, reachable node has an object, then the object may not be available. Nevertheless, we could, at least in principle, still have each node rely only on itself for read availability and durability: nodes could propagate updates and values, and all servers and all clients could store all values. However, fully replicating all data is not appealing for many cloud storage applications.

Depot copes with these limits in two ways. First, Depot provides guarantees on read availability and durability that minimize the required number of correct nodes. Second, Depot makes it likely that this number of correct nodes actually exists. The guarantees are as follows (note that durability—roughly, "the system does not permanently lose my data"—manifests as a liveness property):

- *Read availability.* If during a sufficiently long synchronous interval any reachable and correct node has an object's value, then a read by a correct node will succeed.
- *Durability.* If any correct *hoarding node*, as defined below, has an object's value, then a read of that object will eventually succeed. That is, an update is durable once its value reaches a correct node that will not prematurely discard it.

A *hoarding node* is a node that stores the value of a version of an object until that version is garbage collected (§5.6). In contrast, a *caching node* may discard a value at any time.

To make it likely that the premise of the guarantees holds—namely that a correct node has the data—Depot does three things. First, its *configuration* replicates data to survive important failure scenarios. All servers usually store values for all updates they receive: except as discussed in the remainder of this subsection, when a client sends an update to a server and when servers transmit updates to other servers, the associated value is included with the update. Additionally, the client that issues an update also stores the associated value, so even if all servers become unavailable, clients can fetch the value from the original writer. Such replication allows the system to handle not only the *routine failure* case where a subset of servers and clients fail and lose data but also the *client disaster* and *cloud disaster* cases where all clients *or* all servers fail [5, 14] or become unavailable [8].

Second, *receipts* allow a node to avoid accepting an insufficiently-replicated update. When a server processes an update and stores the update's value, it signs a receipt and sends the receipt to the other servers. Then, we extend the basic protocol to require that an update carry either (a) a *receipt set* indicating that at least k servers have stored the value or (b) the value, itself.

Thus, in normal operation, servers receive and store updates with values, and clients receive and store updates with receipt sets. However, if over some interval, fewer than k servers are available, clients will instead receive, store, and propagate both updates and values for updates created during this interval. Finally, although servers normally receive updates and values together, there are corner cases where—to avoid violating the *always exchange* property—they must accept an update with only a receipt set. Thus, in the worst case Depot can guarantee only that an object value not stored locally is replicated by the client that created it and by at least k servers.

Third, if a client has an outstanding read for version v, it withholds assent to garbage collect v (§5.6) until the read completes with either v or a newer version.

5.4 Bounded staleness

A client expects that soon after it updates an object, other clients that read the object see the update. The following guarantee codifies this expectation:

- *Bounded staleness.* If correct clients $C1$ and $C2$ have clocks that remain within Δ of a true clock and $C1$ updates an object at time t_0, then by no later than $t_0 + 2T_{ann} + T_{prop} + \Delta$, either (1) the update is observable to $C2$ or (2) $C2$ *suspects* that it has missed an update from $C1$.

T_{ann} and T_{prop} are configuration parameters indicating how often a node announces its liveness and how long propagating such announcements is expected to take; both are typically a few tens of seconds.

Depot uses FJC consistency to guarantee that a client always either knows it has seen all recent updates or *suspects* it has not. Every T_{ann} seconds, each client updates a per-client *beacon object* [43] in each volume with its current physical time. When $C2$ sees that $C1$'s beacon object indicates time t, then $C2$ is guaranteed—by FJC consistency—to see all updates issued by $C1$ before time t. On the other hand, if $C1$'s beacon object does not show a recent time, $C2$ *suspects* that it may not have seen other recent updates by $C1$.

When $C2$ *suspects* that it has missed updates from $C1$, it switches to receiving updates from a different server. If that does not resolve the problem, $C2$ tries to contact $C1$ directly to fetch any missed updates and the updates on which those missed updates depend.

Applications use the above mechanism as follows. If a node *suspects* missing updates, then an application that calls GET has two options. First, GET can return a *warning* that the result might be stale. This option is our default; it provides the *bounded staleness* guarantee above. Alternatively, an application that prefers to trade worse availability for better consistency [26] can retry with different servers and clients, blocking until the local client has received all recent beacons.

Note that a faulty client might fail to update its beacon, making all clients *suspicious* all the time. What, then, are the benefits of this bounded staleness guarantee? First, although Depot is prepared for the worst failures, we expect that it often operates in benign conditions. When clients, servers, and the network operate properly, clients are given an explicit guarantee that they are reading fresh data. Second, when some servers or network paths are faulty, *suspicion* causes clients to fail-over to other communication paths to get recent updates.

Bounded staleness v. FJC. Bounded staleness and FJC consistency are complementary properties in Depot. Without bounded staleness, a faulty server could serve a client an arbitrarily old snapshot of the system's state—and be correct according to FJC. Conversely, bounding staleness without a consistency guarantee (assuming that is even possible; we bound staleness by relying on consistency) is not enough. For engineering reasons, our staleness guarantees are tens of seconds; absent consistency guarantees, applications would get confused because there could be significant periods of time when some updates are visible, but related ones are not.

5.5 Integrity and authorization

Under Depot, no matter how many nodes are faulty, only authorized clients can update a key/value pair in a way that affects correct clients' reads: the protocol requires nodes to sign their updates, and correct nodes reject unauthorized updates.

A natural question is: how does the system know which nodes are authorized to update which objects? Our prototype takes a simple approach. When a volume is created, it is statically configured to associate ranges of lookup keys with specific nodes' public keys. This allows specific clients to write specific subsets of the system's objects, and it prevents servers from modifying the objects that they store on behalf of clients. Implementing more sophisticated approaches to key management [48, 71] is future work. We speculate that Depot's FJC consistency will make it relatively easy to ensure a sensible ordering of policy updates and access control decisions [24, 71].

5.6 Data recovery

Even if a storage system retains a consistent and fresh view of the data written to it, data owners care about end-to-end reliability, and the applications and users above the storage system pose a significant risk. For example, many of the failures listed in the introduction may corrupt or destroy valuable data. Depot does not attempt to distinguish "good" updates from "bad" ones or advance the state of the art in protecting storage systems from bad updates. Depot's FJC consistency does, however, provide a basis for applying many standard defenses. For example, Depot can keep all versions of the objects in a volume, or it can provide a basic laddered backup scheme (all versions of an object kept for a day, daily versions kept for a week, weekly versions kept for a month, and monthly versions kept for a year).

Given FJC consistency, implementing laddered backups is straightforward. Initially, servers retain every update and value that they receive, and clients retain the update and value for every update that they create. Then, servers and clients discard the non-laddered versions by *unanimous consent of clients*. Every day, clients garbage collect a prefix of the system's logs by producing a checkpoint of the system's state (using techniques adopted from Bayou [56]). The checkpoint includes information needed to protect the system's consistency and a *candidate discard list* (CDL) that states which prior checkpoints and which versions of which objects may be discarded. The job of proposing the checkpoint rotates over the clients each day.

The keys to correctness here are (a) a correct client will not sign a CDL that would delete a checkpoint prematurely and (b) a correct node discards a checkpoint or version if and only if it is listed in a CDL signed by *all* clients. These checks ensure the following property:

- *Valid discard.* If at least one client is correct, a correct node will never discard a checkpoint or a version of an object required by the backup ladder.

Note that a faulty client cannot cause the system to discard data that it needs: the above approach provides the same read availability and durability guarantees for backup versions as for the current version (§5.3). However, a faulty client can delay garbage collection. If a checkpoint fails to garner unanimous consent, clients notify an administrator, who troubleshoots the faulty client or, if all else fails, replaces it with a new machine. Thus, faulty clients can cause the system to consume extra storage—but only temporarily, assuming that unresponsive clients are eventually repaired or replaced (§3.3).

5.7 Evicting faulty nodes

Depot evicts nodes that provably deviate from the protocol (e.g., by issuing forking writes) and ensures:

- *Valid eviction.* No correct node is ever evicted.

For space, we discuss eviction only at a high level; details are in our technical report [45]. We use proofs of misbehavior (POMs): because nodes' updates are signed, many misbehaviors are provable as such. For example, when a node N observes forking writes from a faulty node M, it creates a POM and slots the POM into the update log, ensuring that the POM will propagate. Note that eviction occurs only if there is a true proof of misbehavior. If a faulty node is merely unresponsive, that is handled exactly as SLA violations are today.

6 Implementation

Our Depot prototype is implemented in Java. It keeps every version written so does not implement laddered backups or garbage collection (§5.6). It is otherwise complete (but not optimized). The prototype uses Berkeley DB (BDB) for local storage and does so synchronously: after writing to BDB, Depot calls *commit* before returning to the caller, and we configure BDB to call `fsync` on every commit.[4]

Implementation of GET & PUT. Depot clients expose a PUT and GET API and implement these calls over the log exchange protocol (§4). Recall that Depot separates data from metadata and that an *update* is only the metadata. Each client node chooses a (usually nearby) *primary server* and fetches updates via background gossip.

On a PUT, a client first locally stores the update and value. As an optimization, rather than initiate the log exchange protocol, a client just sends the update and value of each PUT directly to its primary server. If the update passes all consistency checks and the value matches the hash in the update, the server adds these items to its log and checkpoint. Otherwise, the client and server fall back on log exchange. Similarly, servers send updates and bodies to each other "out of band" as they are received; if two servers detect that they are out of sync, they fall back on log exchange.

On a GET, a client sends the requested lookup key, k, to its primary server along with a *staleness hint*. The staleness hint is a set of two-byte digests, one per logically latest update of k that the client has received via background gossip; note that unless there are concurrent updates to k, the staleness hint contains one element. If the staleness hint matches the latest updates known to the server, the server responds with the corresponding values. The client then checks that these values correspond to the $H(value)$ entries in the previously received updates. If so, the client returns the values to the application, completing the GET. If the server rejects the staleness hint or if the values do not match, then the client initiates a value and update transfer by sending to its primary server (a) its version vector and (b) k. The server replies with (a) the missing updates, which the client verifies (§4.2), and (b) the most recent set of values for k.

If a client cannot reach its primary server, it randomly selects another server (and does likewise if it cannot reach that server). If no servers are available, the client enters "client-to-client mode" for a configurable length of time, during which it gossips with the other clients. In this mode, on a PUT, the client responds to the application as soon as the data reaches the local store. On a GET, the client fetches the values from the clients that created the latest known updates of the desired key.

[4] This approach aids, but does not quite guarantee, persistence of committed data: "synchronous" disk writes in today's systems do not always push data all the way to the disk's platter [52]. Note that if a node commits data and subsequently loses it because of an ill-timed crash, Depot handles that case as it does with any other faulty node.

Depot's overheads are modest. E.g., for 10KB requests 99%-tile latency for GET falls from 2.1 ms to 1.6 ms; for PUT it increases from 14.8 ms to 27.7 ms.	§7.1
Depot imposes little additional cost for read-mostly workloads. For example, Depot's weighted dollar cost of 10KB GETs and PUTs are 2% and 56% higher than the baseline.	§7.2
Depot continues correct operation when failures occur with little impact on latency or resource consumption.	§7.3

FIG. 3—Summary of main evaluation results.

Baseline	Clients trust the server to handle their PUTs and GETs correctly. Clients neither maintain local state nor perform checks on returned data.
B+Hash	Clients attach SHA-256 hashes to the values that they PUT and verify these hashes on GETs.
B+H+Sig	Clients sign the values that they PUT and verify these signatures on GETs.
B+H+S+Store	The same checks as B+H+Sig, plus clients locally store the values that they PUT, for durability and availability despite server failures.

FIG. 4—Baseline variants whose costs we compare to Depot's.

7 Experimental evaluation

The principal question that drives our evaluation is: what is the "price of distrust?" That is, how much do Depot's guarantees cost, relative to the costs of a baseline storage system? We measure latency, network traffic, storage at both clients and servers, and CPU cycles consumed at both clients and servers (§7.1). We then convert the resource overheads into a common currency [29] using a cost model loosely based on the prices charged by today's storage and compute services (§7.2). We then move from "stick" to "carrot", illustrating Depot's end-to-end guarantees (§7.3). Figure 3 summarizes our results.

Method and environment Most of our experiments compare our Depot implementation to a set of *baseline* key-value storage systems, described in Figure 4. All of them replicate the key-value pairs to a set of servers, us-

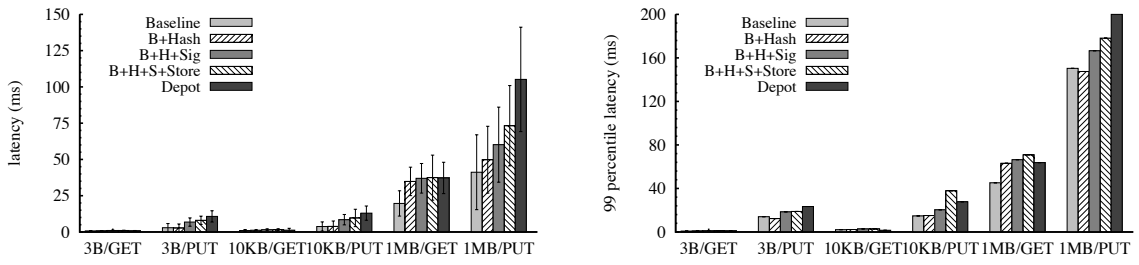


FIG. 5—Latencies ((a) mean and standard deviation and (b) 99th percentile) for GETs and PUTs for various object sizes in Depot and the four baseline variants. For small- and medium-sized requests, Depot introduces negligible GET latency and sizeable latency on PUTs, the extra overhead coming from signing, synchronously storing a local copy, and Depot's additional checks. For large requests, collision-resistant hashing adds significant latency to both PUTs and GETs.

ing version vectors to detect precedence, but omit one or more of Depot's safeguards. In none of the variants do clients check version vectors or maintain history hashes. We have implemented these baseline variants using the same code base as Depot, so they are not heavily optimized. For example, as in Depot, the baselines separate data from metadata, causing writes to two Berkeley DB tables on every PUT, which is possibly inefficient compared to a production storage system. Such inefficiencies may lead to our underestimating Depot's overhead.

Our default configuration is as follows. There are 8 clients and 4 servers with the servers connected in a mesh and two clients connecting to each server. Servers gossip with each other once per second; a client gossips with its primary server every 5 seconds. We experiment with a slightly older implementation that runs without receipts (§5.3) and beaconing (§5.4).

Our default workload is as follows. Clients issue a sequence of PUTs and GETs against a volume preloaded with 1000 key-value pairs. We partition the write key set into several non-overlapping ranges, one for each client. As a result, a GET returns a single value, never a set. A client chooses write keys randomly from its write key range and read keys randomly from the entire volume. We fix the key size at 32 bytes. In each run, each client issues 600 requests at roughly one request per second. We examine three different value sizes (3 bytes, 10 KB, and 1 MB) and the following read-write percentages: 0/100, 10/90, 50/50, 90/10, and 100/0. (We do not report the 10/90 and 90/10 results; their results are consistent with, and can be predicted by, those from the other workloads).

We use a local Emulab [70]. All hosts run Linux FC 8 (version 2.6.25.14-69) and are Dell PowerEdge r200 servers, each with a quad-core Intel Xeon X3220 2.40 GHz processor, 8 GB of RAM, two 7200RPM local disks, and one 1 Gigabit Ethernet port.

7.1 Overhead of Depot

Latency To evaluate latencies in Depot and the baseline systems, we measure from the point of view of the application, from when it invokes GET or PUT at the local library until that call returns. Note that for a PUT, the client commits the PUT locally (if it is a Depot or B+H+S+Store client) and only then contacts the server, which replies only after committing the PUT. We report means, standard deviations, and 99th percentiles, from the GET (i.e., 100/0) and PUT (i.e., 0/100) workloads.

Figure 5 depicts the results. For the GET runs, the difference in means between Baseline and B+Hash are 0.0, 0.2, and 15.2 ms for 3B, 10KB, and 1MB, respectively, which are explained by our measurements of mean SHA-256 latencies in the cryptographic library that Depot uses: 0.1, 0.2, and 15.7 ms for those object sizes. Similarly, the means of RSA-Verify operations explain the difference between B+Hash and B+H+Sign for 3B and 10KB, but not for 1MB; we still investigating overheads for that latter case. Note that Depot's GET latency is lower than that of the strongest two baselines. The reason is that Depot clients verify signatures in the background, whereas the baseline variants do so on the critical path. A key observation is that, for GETs, Depot does not introduce much latency beyond applying a collision-resistant hash to data stored in an SSP—which prudent applications likely do anyway.

For PUTs, the latency is higher. Each step from B+Hash to B+H+S to B+H+S+Store to Depot adds significantly to mean latency, and for large requests, going from Baseline to B+Hash does as well. For example, the mean latency for 10KB PUTs ascends 3.8 ms, 3.9 ms, 8.5 ms, 9.7 ms, 13.0ms as we step through the systems; 99%-tile latency goes 14.8 ms, 15.1 ms, 20.4 ms, 37.9 ms, 27.8 ms.

We can explain the observed Depot PUT latency with a simple model. Depot handles PUTs serially, so we sum the overheads of a PUT's components (microbenchmark means for 10KB are in parentheses, with estimates denoted $\approx$): the client hashes the value (0.2 ms), hashes history ($\approx$ 0.1 ms), signs the update (4.2 ms), stores the body (2.6 ms, with the DB cache enabled), stores the update ($\approx$ 1.5 ms), and transfers the update and body over

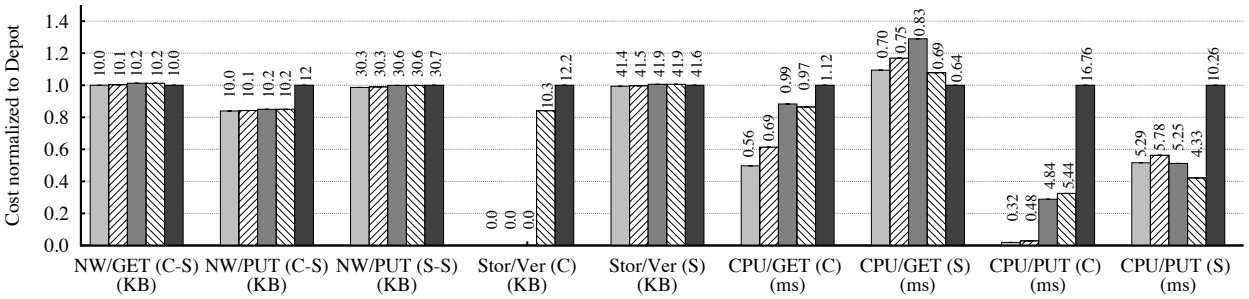


FIG. 6—Resource use of Baseline, B+Hash, B+H+Sig, B+H+S+Store, and Depot. The bar heights represent resource use normalized to Depot, for 10 KB objects and the 100/0 and 0/100 workloads. The labels indicate the actual values. (C) and (S) indicate the average per-request resource use at clients and servers, respectively. (C-S) and (C-S) are client-server and server-server network use, respectively. For storage costs (labeled Stor/Ver), we report the cost of storing a version of an object.

the 1 Gbps network ($\approx$ 0.1 ms); the server verifies the signature (0.3 ms), hashes the value (0.2 ms), hashes history ($\approx$ 0.1 ms), and stores the body (2.6 ms) and update ($\approx$ 1.5 ms). The sum of the means (13.4 ms) is close to the observed latency of 13.0 ms. The model is similarly accurate for the 3B experiments but off by 20% for 1MB; we hypothesize that the divergence stems from queues that build in front of BDB during periodic log exchange.

These PUT latencies could be reduced. For example, we have not exploited obvious pipelining opportunities. Also, we experiment on a 1Gbit/s LAN; in many cloud storage deployments, WAN delays would dominate latencies, shrinking Depot's percentage overhead.

Resource utilization Figure 6 depicts the overheads of various resources in the experiments run above. Depot's overheads are small for GET bandwidth and CPU, for server-server bandwidth, and for server storage cost. The PUT client-server bandwidth overheads are about 20%. The PUT client CPU overheads are substantial due to the additional Berkeley DB access and cryptographic checks. Client storage overheads are also substantial due to the added requirement that clients store data for the PUTs that they create and metadata for all PUTs.

7.2 Dollar cost

Different resources have different costs. To characterize Depot's overall cost, we convert the measured overheads from the prior subsection into dollars. We use the following cost model, loosely based on what customers pay to use existing cloud storage and compute resources.

Client-server network bandwidth	$.10/GB
Server-server network bandwidth	$.01/GB
Disk storage (one client or server)	$.025/GB per month
CPU processing (client or server)	$.10 per hour

Figure 7 shows the overheads from Figure 6 weighted by these costs. Depot's overheads are modest for read-mostly workloads. Depot's GET costs are only slightly higher than Baseline's: \$108.10 v. \$106.50 for 10^8 GET operations on 10KB objects. However, Depot's PUT costs are over 50% higher: \$234.40 v. \$150.50 for 10^8 operations on 10KB objects. Most of the extra cost is from distributing and verifying metadata across all nodes, so the relative overheads would fall for larger objects. Depot's storage costs are 31% higher than Baseline's: \$138.50 v. \$105.50 to store 10^8 10KB objects for a month. Most of the extra cost is from storing a copy of each object at the issuing client; the rest is from storing metadata.

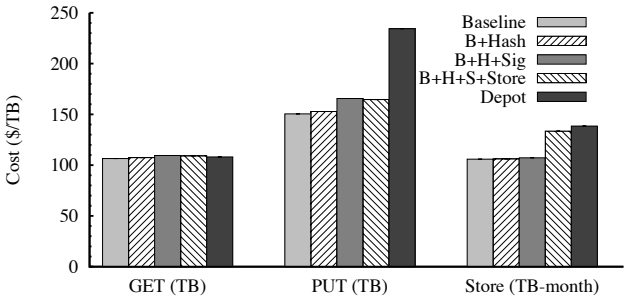


FIG. 7—Dollar cost to GET 1TB of data, PUT 1TB of data, or store 1TB of data for 1 month. Each object has a small key and a 10KB value. 1TB of PUTs or GETs corresponds to 10^8 operations, and 1TB of storage corresponds to 10^8 objects.

7.3 Experiments with faults

We now examine Depot's behavior when servers become unavailable and when clients create forking writes.

Server unavailability In this experiment, 8 clients access 8 objects on 4 servers. The objects are 10KB, and the workload is 50/50 GET/PUT. Servers gossip with random servers every second, and clients gossip with their chosen partner (initially a server) every 5 seconds. 300 seconds into the experiment, we stop all servers. By post-processing logs, we measure the *staleness* of GET results, compared to instantaneous propagation of all updates: the staleness of a GET's result is the time since that result was overwritten by a later PUT. If the GET returns the most recent update, the staleness is 0.

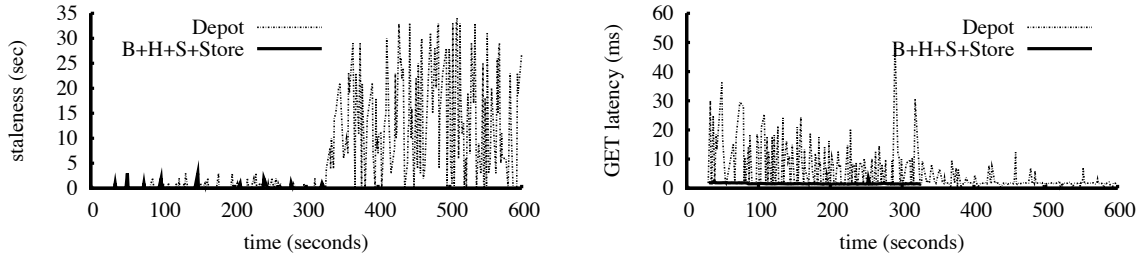


FIG. 8—The effect of total server failure ($t = 300$) on (a) staleness and (b) latency. The workload is 50/50 R/W and 10KB objects. For space, we omit the graph of PUT latency for this experiment. Depot maintains availability through client-to-client transfers whereas the baseline system blocks, and GET latency actually improves (at the expense of staleness).

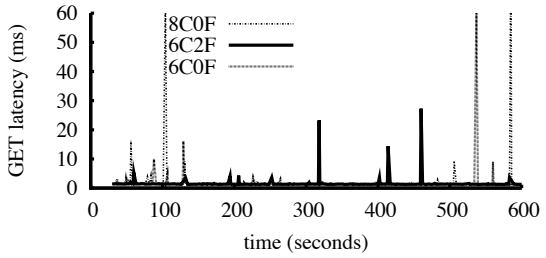


FIG. 9—GET latency seen by a correct client in three runs: 8 correct clients (8C0F), 6 correct clients and 2 faulty clients (6C2F), and 6 correct clients (6C0F). The results for PUT latency are not depicted but are the same: Depot survives forks without affecting client-perceived latency.

Figure 8(a) depicts the staleness observed at one client. Before the servers fail, GETs in both Depot and B+H+S+Store have low staleness. After the failure, B+H+S+Store blocks forever. Depot, however, switches to client-to-client mode, continuing to service requests. Staleness increases noticeably both because it takes more network hops to disseminate updates and because the lower gossip frequency increases the delay between hops.

Figure 8(b) depicts the latency of GETs observed by the same client. Prior to the failure, Depot's GET latency is significantly higher than measured in the experiments in §7.1 because the workload here has just 8 objects, each of which is updated every 2 seconds, so the optimization described in §6 often fails, forcing the client and server to perform a log exchange before the GET can complete. When the servers fail, Depot continues to function, and GET latency actually improves: rather than requesting "the current" value from the server (and then completing a log exchange to get the new metadata required to validate the newest update), in client-to-client mode, a client fetches the specific version mentioned in the update it already has from the writer. Though not depicted, Depot's PUT latency also improves in client-to-client mode: PUT operations return as soon as the update and value are stored locally, with no round trip to a server.

Client fork In this experiment, 8 correct clients (8C0F), 6 correct clients and 2 faulty clients (6C2F), and 6 correct clients (6C0F) access 1000 objects on 4 servers. The objects are 10KB, and the workload is 50/50 GET/PUT. 300 seconds into the experiment, faulty clients begin to issue forking writes. When a correct client observes a fork, it creates and publishes a proof of misbehavior (POM) against the faulty client, and when servers or other clients receive the POM, they stop accepting new writes directly from the faulty client.

Figure 9 depicts the results for GETs. Forks introduced by faulty clients do not have obvious effect on GET or PUT latency; note that the spikes in GET latency prior to $t = 300$ are unrelated to client failures. We also measured CPU consumption and found no interesting differences among the intervals before the failures, at the time of the failures, or after the faulty nodes had been evicted.

8 Teapot for legacy SSPs

Depot runs on both clients and SSP nodes, but it would be desirable to provide Depot's guarantees using unmodified legacy SSPs such as S3, Azure Storage, or Google Storage. Intuitively, such an approach appears possible. In Depot, servers must (1) propagate updates among clients and (2) provide update bodies (i.e., values) in response to GET requests. We should be able to use an SSP's abstract key-value map as a communication channel and as storage for update bodies. And because Depot clients verify everything that they receive from servers, we should still be able to provide most of the properties discussed in §5. In this section, we give a brief overview of *Teapot*, a variation of Depot that uses legacy SSPs.

Teapot assumes an API like that of S3: $\text{LPUT}(k, v, b)$ (associate v with k in a bucket b owned by a given client) and $\text{LGET}(k, b)$ (return v). On a PUT, the Teapot client creates and locally stores the metadata u (a Depot update) and the data d (a Depot value). The client then stores both to the SSP by calling $\text{LPUT}(H(u), u, b_c)$ and $\text{LPUT}(H(d), d, b_c)$, where b_c is a bucket that only c can write. The client then identifies its latest update by storing it to a distinguished key, k_c^* (that is, the client exe-

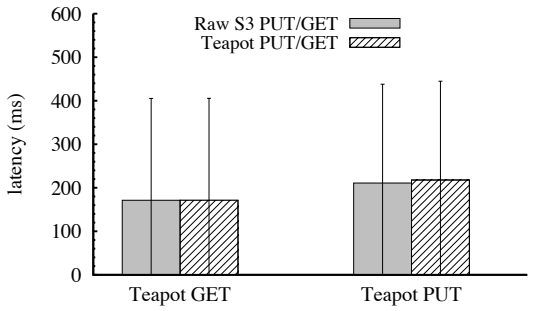


FIG. 10—Average latencies (with standard deviations) perceived by Teapot for GET and PUT operations with 10KB payload when using Amazon S3 for storage.

cutes LPUT(k_c^*, u, b_c)). In the background, the client periodically fetches the other clients' latest updates by reading their k_c^* entries and then fetching and validating the updates' dependencies. On a GET, the Teapot client uses LGET to retrieve the value(s) associated with the latest update(s) that it has received.

We have prototyped Teapot using S3 and a variation on the arrangement just sketched. As shown in Figure 10, accessing S3 through Teapot rather than through LPUT and LGET introduces little latency over S3; the baseline latencies to S3 are already scores of milliseconds, so the additional overheads are small. The other resource costs (client-side storage, extra bandwidth, etc.) are similar to those of Depot (§7.1).

Discussion Teapot differs from Depot in two important ways. First, if a client fails in particular ways, Teapot cannot guarantee *valid discard* (§5.6). A client c can, for example, issue a PUT, allow the update to be observed by other clients, and then delete the value associated with the update. Second, Teapot servers cannot provide the durability receipts that Depot clients use to avoid depending on insufficiently-replicated data (§5.3). Note that Teapot tolerates arbitrary SSP failures and many other client failures (crashes, forks, etc.), so Teapot's additional vulnerability over Depot is limited and may be justified by its deployability.

We now ask: what incremental extensions to SSPs would allow us to run code only on clients but recover Depot's full guarantees? We speculate that the following suffices. First, to allow a correct client to avoid depending on updates that a faulty client could delete, the SSP could implement LINK$(K, b_c, b_{c'})$, UNLINK$(k, b_c, b_{c'})$, and VERIFY(k, H, b_c). LINK causes every existing or new key/value pair in a keyrange K in one client's bucket (b_c) to be *linked to* another client's bucket ($b_{c'}$), where a key/value pair *linked to* another bucket may not be modified or deleted. UNLINK removes such a link. VERIFY checks that the SSP stores a value with hash H for key k in bucket b_c. Then, if a client links to other clients' buckets when it joins the system and it verifies an update's value before accepting the update into its history, we can effectively restore *unanimous consent* for garbage collecting versions (§5.6). Second, to assure clients that updates are sufficiently replicated, the SSP could return a receipt in response to LPUT that the clients could use like receipt sets in standard Depot (§5.3). These extensions seem plausible. Others have proposed receipts [38, 58, 62, 74], and the proposed LINK and UNLINK calls have correlates on Unix file systems, suggesting utility beyond Teapot.

This discussion illustrates that clients can use an SSP-supplied key-value map as a black box to recover most of Depot's properties. To recover all of them, the SSP needs to be incrementally augmented not to delete prematurely.

9 Related work

We organize prior work in terms of trade-offs between availability and fault-tolerance.

Restricted fault-tolerance, high availability. A number of systems provide high availability but do not tolerate arbitrary faults. For example, key-value stores in clouds [16, 21, 22] take a pragmatic approach, using system structure and relaxed semantics to provide high availability. Also, systems like Bayou [67], Ficus [61], PRACTI [10], and Cimbiosys [60] can get high availability by replicating all data to all nodes. Unlike Depot, none of these systems tolerates arbitrary failures.

Medium fault-tolerance, medium availability. Another class of systems provides safety even when only a subset (for example, 2/3 of the nodes) is correct. However, the price for this increased fault tolerance compared to the prior category is decreased liveness and availability: to complete, an operation must reach a quorum of nodes. Such systems include Byzantine-Fault Tolerant (BFT) replicated state machines (see [15, 19, 30, 33]) and Byzantine Quorums [46]. Note that researchers are keenly interested in reducing trust: compared to classic BFT systems, the recently proposed A2M [17], TrInc [42], and BFT2F [44] all tolerate more failures, the former two by assuming trusted hardware and the latter by weakening guarantees. However, unlike Depot, these systems still have fault thresholds, and none works disconnectedly. PeerReview [31] requires a quorum of witnesses with complete information (hindering liveness), one of which must be correct (a trust requirement that Depot does not have).

High fault-tolerance, low availability. In fork-based systems, such as SUNDR [43] and FAUST [12], the server is totally untrusted, yet even under faults provides a safety guarantee: fork-linearizability, fork-sequential consistency, etc. [54]. However, these systems provide reduced liveness and availability compared to Depot.

First, in benign runs, their admittedly stronger semantics means that they cannot be available during a network partition or server failure. Second, after a fork, nodes are "stranded" and cannot talk to each other, effectively stopping the system. A related strand of work focuses on *accountability* and *auditing* (see [38, 58, 62, 74]), providing proofs to participants if other participants misbehave. All of these systems *detect* misbehavior, whereas our aim is to *tolerate* and *recover* from it—which we view as a requirement for availability.

Systems with similar motivations. Venus [63] allows clients not to trust a cloud storage service. While Venus provides consistency semantics stronger than Depot's (causal consistency for pending operations, linearizability for completed operations (roughly)), it makes stronger assumptions than Depot. Specifically, Venus relies on an untrusted verifier in the cloud; assumes that a core set of clients does not permanently go offline; and does not handle faulty clients, such as clients that split history. SPORC [24] is designed for clients to use a single untrusted server to order their operations on a single shared document and provides causal consistency for pending operations (and stronger for committed operations). Unlike Depot, SPORC does not consider faulty clients, allow clients to talk to any server, or support arbitrary failover patterns. However, SPORC provides innate support for confidentiality and access control, whereas Depot layers those on top of the core mechanism.

A number of other systems have sought to minimize trust for safety and liveness. However, they have not given a correctness guarantee under arbitrary faults. For example, Zeno [64] does not operate with maximum liveness or minimal trust assumptions: it assumes $f+1$ available servers per partition, where f is the number of faulty servers. TimeWeave [47] ensures that correct nodes can pass the blame of any mal-activity to culprit nodes, and S2D2 [35] uses tamper-evident history summaries to detect forks. However, unlike Depot, these two systems neither *repair* forks nor target cloud storage (which requires addressing staleness, durability, and recoverability). Other systems target scenarios similar to cloud storage but do not protect consistency [28, 34, 65].

Some systems have, like Depot, been designed to resist large-scale correlated failures. Glacier [32] can tolerate a high threshold, but still no more than this threshold, of faulty nodes, and it stores only immutable objects. OceanStore [39] is designed to minimize trust for durability but does not tolerate nodes that fail perniciously.

Distributed revision control. Distributed repositories like Git [27], Mercurial [49], and Pastwatch [73] incorporate a data model similar to Depot's, and could be augmented to resist faulty nodes (for example, forcing clients to sign updates in Git would prevent servers from undetectably altering history). However, all of these systems are fundamentally geared toward replicating a source code repository. Our context brings concerns that these systems do not address, including how to avoid clients' storing all data, how to perform update exchange in this scenario, how to provide freshness, how to evict faulty nodes, how to garbage collect, etc.

10 Conclusion

Depot began with an attempt to explore a radical point in the design space for cloud storage: *trust no one*. Ultimately we fell short of that goal: unless all nodes store a full copy of the data, then nodes must rely on one another for durability and availability. Nonetheless, we believe that Depot significantly expands the boundary of the possible by demonstrating how to build a storage system that eliminates trust assumptions for safety and minimizes trust assumptions for liveness.

Acknowledgments

Insightful comments by Marcos K. Aguilera, Hari Balakrishnan, Brad Karp, David Mazières, Arun Seehra, Jessica Wilson, the anonymous reviewers, and our shepherd, Michael Freedman, improved this paper. The Emulab staff was a great help, as always. This work was supported by ONR grant N00014-09-10757, AFOSR grant FA9550-10-1-0073, and NSF grant CNS-0720649.

References

[1] Amazon Simple Storage Service (Amazon S3). `http://aws.amazon.com/s3`.

[2] AWS forum: Customer app catalog. `http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=66`.

[3] Google storage for developers. `http://code.google.com/apis/storage/docs/overview.html`.

[4] Windows Azure Platform. `http://www.microsoft.com/windowsazure/windowsazure`.

[5] Victims of lost files out of luck. `http://news.cnet.com/Victims-of-lost-files-out-of-luck/2100-1023_3-887849.html`, Apr. 2002.

[6] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: a case for cloud storage diversity. In *Proc. 1st ACM Symp. on Cloud Comp.*, 2010.

[7] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9(1):37–49, 1995.

[8] Amazon S3 Team. Amazon S3 availability event: July 20, 2008. `http://status.aws.amazon.com/s3-20080720.html`, July 2008.

[9] C. Beckmann. Google app engine: Information regarding 2 July 2009 outage. `http://groups.google.com/group/google-appengine/browse_thread/thread/e9237fc7b0aa7df5/ba95ded980c8c179`, July 2009.

[10] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI*, 2006.

[11] A. Birrell, R. Levin, R. Needham, and M. Schroeder. Grapevine: An Exercise in Distributed Computing. *CACM*, 25(4), 1982.

[12] C. Cachin, I. Keidar, and A. Shraer. Fail-Aware Untrusted Storage. In *DSN*, 2009.

[13] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *PODC*, 2007.

[14] M. Calore. Ma.gnolia suffers major data loss, site taken offline. In *Wired*, Jan. 2009.

[15] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4), 2002.

[16] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[17] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested Append-Only Memory: Making Adversaries Stick to their Word. In *SOSP*, 2007.

[18] CircleID. Survey: Cloud computing 'no hype', but fear of security and control slowing adoption. `http://www.circleid.com/posts/20090226_cloud_computing_hype_security/`, Feb. 2009.

[19] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché. UpRight cluster services. In *SOSP*, 2009.

[20] B. Cook. Seattle data center fire knocks out Bing Travel, other web sites. `http://www.techflash.com/seattle/2009/07/Seattle_data_center_fire_knocks_out_Bing_Travel_other_Web_sites_49876777.html`, July 2009.

[21] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *VLDB*, 2008.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[23] M. Demmer, B. Du, and E. Brewer. TierStore: a distributed filesystem for challenged networks in developing regions. In *FAST*, 2008.

[24] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *OSDI*, Oct. 2010.

[25] M. Frigo and V. Luchangco. Computation-Centric Memory Models. In *SPAA*, 1998.

[26] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of Consistent, Available, Partition-tolerant web services. In *ACM SIGACT News, 33(2)*, 2002.

[27] Git: The fast version control system. `http://git-scm.com/`.

[28] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Network and Distributed System Security (NDSS) Symposium*. Internet Society (ISOC), 2003.

[29] J. Gray and P. Shenoy. Rules of Thumb in Data Engineering. In *Data Engineering*, pages 3–12, 2000.

[30] R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. The next 700 BFT protocols. In *Eurosys*, 2010.

[31] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *SOSP*, 2007.

[32] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *NSDI*, 2005.

[33] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-Overhead Byzantine Fault-Tolerant Storage. In *SOSP*, 2007.

[34] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Conference on File and Storage Technologies (FAST)*, 2003.

[35] B. Kang. *S2D2: A framework for scalable and secure optimistic replication*. PhD thesis, UC Berkeley, Oct. 2004.

[36] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC*, 1997.

[37] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM TOCS*, 10(1):3–5, Feb. 1992.

[38] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A durable and practical storage system. In *USENIX Technical*, 2007.

[39] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *ASPLOS*, 2000.

[40] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7), July 1978.

[41] L. Lamport, R. Shostack, and M. Pease. The Byzantine Generals Problem. *ACM TPLS*, 4(3):382–401, 1982.

[42] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: small trusted hardware for large distributed systems. In *NSDI*, 2009.

[43] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[44] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, 2007.

[45] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust (extended version). Technical Report TR-10-33, UT Austin, Sept. 2010.

[46] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11(4):203–213, Oct. 1998.

[47] P. Maniatis. *Historic Integrity in Distributed Systems*. PhD thesis, Stanford, 2003.

[48] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *SOSP*, 1999.

[49] Mercurial. `http://mercurial.selenic.com/`.

[50] R. Miller. FBI siezes servers at Dallas data center. `http://www.datacenterknowledge.com/archives/2009/04/03/fbi-seizes-servers-at-dallas-data-center/`, Apr. 2009.

[51] S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Subtleties in Tolerating Correlated Failures in Wide-area Storage Systems. In *NSDI*, 2006.

[52] E. Nightingale, K. Veeraraghavan, P. Chen, and J. Flinn. Rethink the sync. *ACM TOCS*, 26(3), 2008.

[53] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *USITS*, 2003.

[54] A. Oprea and M. Reiter. On consistency of encrypted files. In *DISC*, 2006.

[55] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, S. Kiser, D. Edwards, and C. Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE TSE*, 9(3):240–247, May 1983.

[56] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *SOSP*, 1997.

[57] E. Pinheiro, W. Weber, and L. Barroso. Failure trends in a large disk drive population. In *FAST*, Feb. 2007.

[58] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with CloudProof. Technical Report MSR-TR-2010-46, Microsoft Research, May 2010.

[59] V. Prabhakaran, L. Bairavasundaram, N. Agrawal, H. Gunawi, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. IRON file systems. In *SOSP*, 2005.

[60] V. Ramasubramanian, T. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *NSDI*, 2009.

[61] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving File Conflicts in the Ficus File System. In *USENIX Summer*, 1994.

[62] M. Shah, M. Baker, J. Mogul, and R. Swaminathan. Auditing to keep online storage services honest. In *HotOS* , 2007.

[63] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *CCSW*, Oct. 2010.

[64] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent Byzantine fault tolerance. In *NSDI*, Apr. 2009.

[65] J. Strunk, G. Goodson, M. Scheinholtz, C. Soules, and G. Ganger. Self-securing storage: protecting data in compromised systems. In *OSDI*, 2000.

[66] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *ICPDS*, 1994.

[67] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[68] US Secret Service report on insider attacks. `http://www.sei.cmu.edu/about/press/insider-2005.html`, 2005.

[69] W. Vogels. Life is not a state-machine: The long road from research to production. In *PODC*, 2006.

[70] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, Dec. 2002.

[71] T. Wobber, T. L. Rodeheffer, and D. B. Terry. Policy-based access control for weakly consistent replication. In *EuroSys*, 2010.

[72] J. Yang, C. Sar, and D. Engler. EXPLODE: A lightweight, general system for finding serious storage system errors. In *OSDI*, 2006.

[73] A. Yip, B. Chen, and R. Morris. Pastwatch: A distributed version control system. In *NSDI*, 2006.

[74] A. Yumerefendi and J. Chase. Strong accountability for network storage. *ACM Transactions on Storage (TOS)*, 3(3), Oct. 2007.

Comet: An active distributed key-value store

Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno,
Arvind Krishnamurthy, Henry M. Levy
University of Washington

Abstract

Distributed key-value storage systems are widely used in corporations and across the Internet. Our research seeks to greatly expand the application space for key-value storage systems through *application-specific customization*. We designed and implemented Comet, an extensible, distributed key-value store. Each Comet node stores a collection of *active storage objects* (ASOs) that consist of a key, a value, and a set of *handlers*. Comet handlers run as a result of timers or storage operations, such as `get` or `put`, allowing an ASO to take dynamic, application-specific actions to customize its behavior. Handlers are written in a simple sandboxed extension language, providing properties of safety and isolation.

We implemented a Comet prototype for the Vuze DHT, deployed Comet nodes on Vuze from PlanetLab, and built and evaluated over a dozen Comet applications. Our experience demonstrates that simple, safe, and restricted extensibility can significantly increase the power and range of applications that can run on distributed active storage systems. This approach facilitates the sharing of a single storage system by applications with diverse needs, allowing them to reap the consolidation benefits inherent in today's massive clouds.

1 Introduction

The last decade has seen the rise of distributed storage systems built on loosely coupled collections of autonomous computers. For example, Amazon's S3 [3] provides a key-value storage service for external Web clients. Amazon's Dynamo [17], Apache Cassandra [5], and Project Voldemort [38] provide reliable and scalable key-value stores for company-internal applications (for Amazon, Facebook, and LinkedIn, respectively). On the global Internet, DHTs provided by BitTorrent-based systems, such as Vuze [58] and uTorrent [56], store metadata for millions of clients using peer-to-peer file-sharing applications. And finally, researchers have developed complete file systems on top of untrusted clients in widely distributed P2P environments [2, 14, 44].

Distributed storage systems offer many advantages over their centralized counterparts. For example, a decentralized structure supports scalability; the lack of centralized management enhances automatic load balancing; and the use of replication in a highly distributed environment can improve reliability and data availability. We therefore expect Dynamo-like storage systems to become commonplace as generic application infrastructures in the future, both inside of the enterprise and as shared services on the Internet.

A significant limitation of such systems for generic application support, however, is that different applications have different needs. As an example, each Dynamo application inside of Amazon runs its own Dynamo instance [17], even though a single instance might be logically better and more resource efficient. In our own work on Vanish [25] – a security-oriented DHT application – we needed to make application-specific parameter and policy changes to Vuze (a million-node commercial DHT) in order to harden it against attack. While these changes were conceptually simple, e.g., modifying the storage replication algorithm, deploying our changes took months of work with Vuze's DHT designer. Other Vuze applications may wish to make their own application-specific changes or enhancements, but doing so is neither feasible nor supportable, and it doesn't scale. We believe that with the huge consolidation benefits of shared cloud storage services, either inside or outside of the enterprise, supporting specialization of storage services can have high payoffs in the future.

This paper presents Comet, a next-generation, flexible, distributed storage system, which opens the world of distributed storage to a new set of more complex storage applications. In particular, Comet permits multiple applications to share a single Comet instance, while enabling each application to change the behavior of its storage elements to suit its own requirements. For example, a storage element can make decisions based on its access history, its current number of replicas, the time of day, etc. Therefore Comet can easily support different storage lifetimes, access methods, access control schemes, or replication schemes for different storage-element types, in a way that makes them easy to deploy and test. Using Comet, we can also carry out interesting measurement-based experiments from *within* the DHT.

Comet implements *active storage objects* (ASOs). An active storage object consists of a key, an associated value (an untyped blob), and optionally, a set of simple *handlers*. An ASO's handlers execute as a result of common storage events on the object (such as `get` and `put`) or from timer events that its handlers request. As a result, an ASO can modify its environment, monitor its execution, and make dynamic decisions about its state.

The design of an extensible system for this environment presents a set of interesting design questions. For example, what features should the system provide for ap-

plications and which can (and should) be left out? What is the proper tradeoff between power and safety? How can client nodes be confident that active storage objects will not cause damage or interference? How can we prevent the use of active storage objects to mount a DDoS attack? And overall, how can we extend the storage system without losing its principal characteristics? Our Comet design considers these and other issues.

The remainder of this paper describes our goals, architecture, experience, and evaluation of Comet. To provide concrete insight into Comet's design and potential, we implemented a Comet prototype and used it to create and deploy a set of over a dozen Comet applications. Our prototype leverages Vuze: each Comet instance is an extended Vuze client that can execute Comet active storage objects while also serving as a full participant in the million-node Vuze DHT. Comet applications are written in Lua – a common application-extension language. We modified the Lua runtime to meet our isolation and safety requirements, providing a safe sandbox for handler execution. To test our applications we ran our Comet clients from several hundred PlanetLab nodes and measured their behavior. Overall, our experience demonstrates that a highly restrictive but active distributed storage system can provide significant power to simultaneously support applications with diverse storage needs.

2 Related Work

The concept of extensible systems has been widely explored in the past in several domains. Extensible operating systems have been proposed that support application-specific needs [6, 46, 28]. Active networks allow code to be downloaded along with network data and executed within the network infrastructure (e.g., on routers) to extend network services [60, 54]. Active messages execute a small amount of user code with each message reception [57]. Click explored the design of an extensible router [30]. Database triggers allow applications to define procedural code that is executed in response to database operations [35].

In the context of storage systems, Watchdogs [7] extends the Unix file system, allowing a user-mode process to interpose on file operations for specific files to change access semantics. Several projects have proposed the integration of CPUs and disks to create intelligent disk storage systems that can provide on-board application-specific functions, e.g., for decision support systems, data mining, and image processing [29, 41, 1].

DHTs are increasingly used to support a variety of distributed applications, such as file-sharing, distributed resource tracking, end-system multicast, publish-subscribe systems, distributed search engines, and even data-center applications. Some of these systems (e.g., as CFS [14], i3 [52], and PAST [44]) can be implemented using the traditional put/get interface, but many others (e.g., Mercury [8], CoralCDN [21], Scribe [45], and Bayeux [64]) require customized interfaces and are implemented by altering the underlying DHT mechanisms in significant ways. Our work provides the ability to extend a DHT without requiring a substantial investment of effort to modify its implementation.

Deployed DHTs don't currently offer good semantics and security. However, people do know how to make them consistent [32, 34] and harden then against attacks [11, 16, 48, 26, 59]. The reason DHTs do not currently implement these techniques is that there has not yet been a deployed application that truly needed strong semantics and security. For example, the Vuze design perceived many threats as irrelevant [23] and deployed few defenses against them. However, after the new, more demanding Vanish application was proposed [25], the Vuze DHT responded by embracing a variety of effective security measures. In addition to enabling new applications atop DHTs, we hope to drive the design of these systems towards well-understood, yet unadopted levels of security and consistency.

3 Goals

Comet is a distributed key-value storage system. Like other such systems, a Comet storage object is a <key,value> pair. Unlike previous systems, however, Comet's design facilitates extensible, active storage objects. A Comet application performing a `put` can therefore include, along with a key and value, a small set of *handlers* for that object. The node receiving the `put` stores the handlers along with the key and value, registers the handlers for events that they specify, and executes the handlers when their respective events occur.

Comet's system goals are:

1. *Flexibility.* Comet should be easily customizable to achieve our target functions described below.
2. *Isolation and safety.* A client node running Comet should be protected from the execution of handlers (e.g., an executing handler cannot corrupt the node or use unlimited resources). Handlers should not be able to mount messaging attacks on other nodes.
3. *Performance.* The performance of `gets/puts` on a Comet ASO with null handlers should be the same as on a non-active system, and execution of handlers should have only negligible performance impact.

Isolation and safety are particularly important to our architecture. While Comet can be used in different environments, we designed it to enable wide-scale, outside-the-firewall deployment on autonomous nodes, similar to P2P systems and DHTs. Users downloading Comet must trust it and have guarantees about its behavior. For this reason, Comet enforces four important restrictions:

1. *Limited knowledge:* an ASO is not aware of other objects or resources stored on the same node and has no direct way to learn about them.
2. *Limited access:* an object handler can manipulate only its own value and cannot modify the values of other objects on its storage node.
3. *Limited communication:* an active storage object cannot send arbitrary messages over the network.
4. *Limited resource consumption:* an ASO's resource usage is strictly bounded, e.g., the system limits the amount of computation and memory it can consume.

We are specifically *not* attempting to build a general-purpose distributed programming system, such as PlanetLab [4, 36]; such a system would be unacceptable in our target environment and inappropriate (and unnecessary) for our needs. Rather, our goal is to support relatively simple specializations or actions on simple storage objects. Even very simple specializations can provide a significantly more powerful storage system that enables new types of applications. We therefore take a lightweight and limited approach. As examples, an ASO should be able to perform the following functions:

- *Statistics gathering.* Collect statistics about its use, e.g., by counting the number of `get`s and `put`s.
- *Information tracking.* Log information, such as a list of IPs that performed `get` operations on its value or a recent history of the values it stored.
- *Time awareness.* Take time-based actions, e.g., to make periodic changes to its state or self-destruct after a timer has elapsed.
- *Location awareness.* Make location-based decisions, e.g., choosing where to store based on nodes' network locations.
- *Access control.* Implement simple access control policies on its own.
- *Replication.* Implement different replication policies.
- *Storage system measurement.* Provide insight into the behavior of the distributed storage system as seen by clients executing within the system itself.

As we shall see, the only long-term state available to a handler is its object's value; therefore, any logs, counts, etc., must be stored as part of that value. However, an active object can choose to report only a subset of its stored value record on a `get`, or it can selectively report different values to different callers based on call parameters.

The following sections describe Comet's architecture. In particular, we discuss the tradeoffs required to provide flexibility while also achieving isolation and safety.

4 Architecture and Implementation

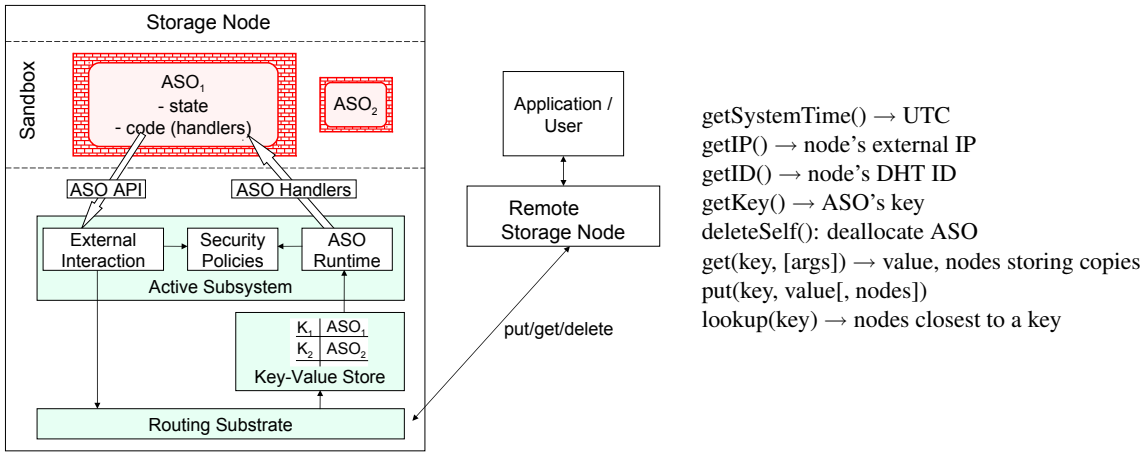
This section describes Comet's active storage architecture and prototype implementation. One could imagine running Comet in various environments, e.g., an inside-the-firewall corporate deployment or a distributed environment with autonomous untrusted nodes. We focus our current architecture and prototype on the latter.

4.1 Architecture

Figure 1(a) shows the high-level architecture of our Comet distributed storage system. The Comet storage system consists of three basic components. First is the *routing substrate* (Figure 1(a) bottom), which implements the value/node mapping, allowing a client to find nodes that store specific data items. In the case of a DHT, for example, the routing substrate typically applies a hash function to the key to compute the IDs of nodes that store the associated value. However, other routing substrates may locate values in other ways.

The second component is the *key-value store*, which maintains a set of key-value pairs on each node. A key-value storage system typically exports a simple `get/put` interface. While existing storage systems store arbitrary, untyped byte strings, the Comet storage system stores *active storage objects* (ASOs). An ASO consists of a key and its associated *state* (i.e., a value, stored as an untyped byte string), along with optional *code* that operates on that state. The code is structured as a set of *handlers* that specify how the object behaves, i.e., how it modifies its state when certain events occur. For example, an ASO's `onGet` handler is invoked whenever a remote client performs a `get` operation to access an object. This handler might perform some simple operation, such as incrementing a counter for the number of gets or appending the client's IP address to a log structure. The counter or the log structure would be stored as part of the ASO's state that can be accessed by the handler.

The third architectural component is the *active runtime system*. The runtime system handles ASO invocations and provides the security policy and execution environment. An application running on a remote client specifies the initial state and handlers for an ASO when initially storing the object via a `put` operation. When a client performs a `get` or a `put`, it can optionally request a cryptographic checksum of the code associated with the target ASO. This can serve as an integrity check that the client's initial `put` is to a key with no associated ASO and that subsequent operations are performed on ASOs created by the application. In most implementations, a Comet node distrusts remote nodes and client applications; therefore, the runtime component of the active subsystem implements and enforces an ASO execution sandbox (Figure 1(a), top). Our Comet prototype uses a language sandbox based on Lua [43] to prevent a handler



(a) Architecture. (b) ASO API.

Figure 1: **Comet Architecture and APIs.** (a) depicts the decomposition of a Comet node into two vertical components - the core Comet code, which is trusted from the node's perspective, and the ASO code which is arbitrary and, therefore, untrusted. (b) details the API exposed to ASOs.

from accessing outside state and to constrain the ASO from consuming too many computational and memory resources on the host. The ASO runtime consults a security policy module, which specifies all execution limits.

While some applications may be satisfied by an entirely sandboxed execution, many would benefit from an ASO's limited ability to interact with or "sense" its environment. For example, to implement the conditional replication scheme we added to Vuze for Vanish, an ASO requires knowledge of the number of replicas in the DHT and the time of day (to enforce the desired minimum replication interval). For this reason, the active subsystem exposes a small API (called the ASO API) to the handlers.

4.2 Active Storage Object API

Table 1 and Figure 1(b) show the handler and ASO runtime APIs, respectively. The handler API supports invocations based on the primary storage functions – `put`, `get` – as well as an `onTimer` handler to be executed periodically (e.g., once every 10 minutes) during the object's lifetime. For example, an ASO could directly implement a custom replication policy in its `onTimer` handler.

The ASO runtime API is the only way for an ASO to interact with its environment outside of the sandbox. Our design supports two types of useful interactions: (1) obtaining information about the local node, and (2) executing various storage system operations. The former category includes functions to obtain the time of day, the hosting machine's external IP address, etc. The latter includes functions to interact with other storage system objects. The ASO API was not designed to be entirely general; rather, our goal was to provide a minimal interface, informed in part by our requirements of security, privacy, and isolation. We tested this interface by implementing and running over a dozen applications on our Comet prototype. Interestingly, we were able to build a relatively diverse set of applications with a surprisingly small interface, which has remained relatively stable through the project. This suggests that a small interface, like the one shown in Figure 1(b), can support a wide variety of applications. Naturally, there are limitations. For example, we explicitly prohibit any direct network-level interactions with remote nodes on the Internet. While this feature might be desirable to certain measurement applications, its DDoS implications would be unacceptable.

onGet(caller[, callbackID, payload]) Invoked when a *get* is performed on the ASO. Returns a value which will be passed back to the caller. Instead of returning a value immediately, the handler could also perform a *put* at the optional *callbackID* sometime in the future. The handler also takes an optional *payload* argument of arbitrary type.
onPut(caller) Invoked upon initial *put* when the object is created. Returns the value that should be stored by the node (e.g., itself or nil).
onUpdate(new_value, caller) Invoked on an ASO when a *put* overwrites an existing value. Returns the value that should be stored, e.g., *new_value* if it should be replaced, or itself if not.
onTimer() Invoked periodically. This handler has no return value. It is used to perform periodic maintenance such as replication.

Table 1: **ASO Handler Calls.**

4.3 Language Based Sandbox

Our Comet prototype focuses on a DHT environment composed of a large number of untrusted autonomous nodes that cooperate to support the distributed active storage system. In this environment, the key challenges include providing a strong sandbox and limiting ASO resource consumption. We briefly describe how our system addresses these challenges using a language based sandbox.

The Comet prototype required an ASO programming environment that reflected our needs for simple extensibility, flexibility, performance, isolation, and safety. To meet these needs, we chose Lua [43], a lightweight and easily constrained scripting language. A dynamically typed, imperative and functional programming language, Lua is most commonly used for coding application extensions. In this context, it lets users add or modify features in video game engines, Web servers, version control systems and other applications (specific examples include World of Warcraft, SimCity 4, Adobe Photoshop Lightroom, and Squeezebox Jive Platform). Several properties make Lua well suited for implementing ASOs. First, it employes a small set of programming constructs (including first-order functions) and a small number of data types (including tables, which are heterogeneous associative arrays). Second, Lua compiles to simple bytecode, which makes it relatively easy to sandbox. Finally, ASOs written in Lua are concise and small when serialized; the Lua ASOs we implemented are all under 1.5KB, about five to ten times smaller than Java equivalents.

Comet represents ASOs as Lua tables that encapsulate both persistent state and the handlers to be invoked on that state. Lua tables can implement basic arrays, associative arrays, or both. While an associative array can contain any name-value mappings, we treat certain associations as handlers. In particular, if the ASO table contains an associative array with the names "onGet," "onPut," "onUpdate," or "onTimer" – and those names are associated with values that are Lua functions – then the runtime invokes those functions when the corresponding events occur. Our runtime system serializes Lua tables into a byte stream for transmission to a storage node on a `put` request.

We made several modifications to the standard Lua interpreter for the Comet runtime system. We sandbox ASOs by removing all but the core libraries from the runtime, leaving only a math package, string manipulation, and table manipulation. As a result, handlers are extremely restricted: they have no direct network access, no system execution capabilities, no thread creation capabilities, and no file system access. We also strictly bound the amount of resources that a handler can consume. For example, the runtime limits both the number of bytecode instructions that a handler can execute and the amount of memory it can consume. If a handler exceeds either of these limits, the runtime terminates its execution.

The Comet runtime exposes a DHT wrapper object to handlers, which allows an ASO to communicate with its environment. The ASO can learn information about the hosting node, including the external IP address and the current system time. It can also perform a restricted set of DHT operations. For example, it can perform `get` and `put` operations on *replicated* copies of its value stored at other nodes. In the API presented in Section 4.2, these operations return values or neighboring node IDs. However, since these operation are slow in the DHT setting and may block for seconds or even minutes, we chose to implement them using function callbacks. Each such operation takes an optional parameter, a function which accepts the result as its parameter. For example, instead of returning a value, a `get` operation takes a function which is eventually passed the result of the operation. The operation returns immediately with no value, and the `get` is actually performed after the ASO execution has completed. While this presents a slightly different paradigm to the user, we think this provides a greater ability to optimize the performance of Comet-based applications.

4.4 Comet Prototype Implementation

We built the Comet prototype on the Vuze DHT, which supports the widely used Vuze BitTorrent client. The DHT is used mainly for distributed tracking of torrents; however it has been used in research as well [27, 25].

Vuze implements the Kademlia routing protocol, in which each node is assigned a 160-bit ID based on the SHA1 hash of its IP address and port. Basic DHT operations (`get`, `put`, and `remove`) take a 160-bit key, perform a lookup to find nodes whose ID is *close* to that key, and then send a read or store RPC to those nodes.

We minimally extended the Vuze interface to conform to Comet's abstract operations. For example, we augmented `get` to allow a caller to pass an arbitrary bytestring argument. This supports a parameterized `get` operation, where the ASO can return different values depending on the parameter (analogous to the semantics for GET in HTTP).

Allowing extensibility in a DHT environment creates challenges, e.g., it has the potential to provide a platform for DDoS attacks. Therefore, in addition to the Lua resource restrictions described previously, we limit DHT communications that ASOs can perform in two ways.

First, we do not allow an ASO to perform operations on arbitrary DHT keys or nodes, but rather only on specific key-node pairs. An ASO may communicate with any of its neighboring nodes that are responsible for replicas of the ASO. We also allow the ASO to communicate with key-node pairs that have interacted with it in the past, once for each such interaction. To enable this functionality, we extended Comet requests to include the ID of the requesting node and the ID of a local key contained within the node. If an ASO receives a `get` request with a key ID specified, it gains the capability for a one-time operation on that key to the node that issued the request. The ASO can then either return a value immediately and exhaust its one-time capability, or save that capability for future use. This mechanism allows applications to respond to DHT requests at a future point in time, es-

pecially if the requested data is not currently available. We do not allow ASOs to pass these capabilities between each other as doing so would enable a malicious node to mount DDoS attacks. In Section 5 we discuss signed ASOs, which do not have these restrictions.

Second, Comet imposes rate limits on the number of messages generated by an ASO, either to neighboring nodes storing replicas or to arbitrary key-node pairs that have interacted with it in the past. This prevents misbehaving ASOs from exhausting the bandwidth resources of the Comet nodes hosting them. We discuss these security issues further in Section 7.

5 Applications

This section seeks to demonstrate both the range of storage behaviors that Comet can support and the ease with which those behaviors can be implemented. To do this, we describe several of the active storage applications we have implemented, deployed, and measured on our Comet PlanetLab prototype. We provide code snippets to show how simply these actions can be programmed in our Lua-based ASO environment. In Section 6, we present measurements from some of these examples.

5.1 Customizable Replication

Most DHTs specify a fixed replication policy for stored values, requiring applications to conform to that policy. In contrast, Comet ASOs can provide their own application-specific replication mechanisms, e.g., controlling the replication factor, the replication interval, and the choice of nodes on which the object will be replicated. This flexibility is useful for applications that place varying degrees of emphasis on performance, availability, locality, and security. For instance, a security sensitive application (such as Vanish) might use a small number of replicas and long replication intervals, limiting the dispersion of its objects stored in the DHT. On the other hand, an application that values availability might replicate frequently to a large number of nodes.

Listing 1 shows how an ASO can define a customized replication policy. In this example, the `onTimer` handler wakes up periodically, invokes `lookup` to determine a list of nodes closest to the ASO's key, executes `selectGoodNodes`[1] to identify a subset of nodes that will serve as replicas, and then stores a copy of itself on the selected nodes using `put`. We have also implemented a timer handler that replicates only when the number of existing replicas falls below a certain threshold; this lowers communication overhead and mitigates data harvesting attacks for security sensitive applications, reflecting the changes we made to Vuze after we published Vanish [25].

[1]The Lua code for `selectGoodNodes` is omitted for brevity. It implements an application-specific policy for choosing replicas.

```
function aso:handleLookup(nodes)
    nodes = self.selectGoodNodes(nodes)
    dht.put(dht.getKey(), self, nodes)
end
function aso:onTimer()
    dht.lookup(dht.getKey(), self.handleLookup)
end
```

Listing 1: Smart Replication

5.2 Controlling Data Access

Comet objects can implement various policies that control how data stored in the objects is accessed. We illustrate a few such examples.

Timeouts and Limited-read values: ASOs can be used to implement objects that will be accessible for only a limited, application-specified time. Such objects are meaningful for security applications such as Vanish [25], which provide support for self-destructing digital data by storing cryptographic keys in a DHT.

Listing 2 shows the handler code required to implement application-specific timeouts. Each replica stores a timestamp when the object is created (stored) and then deletes the object after 60 minutes using a timer handler. In addition, the `onGet` handler prevents the object's contents from being accessed after the timeout but before it is deleted by a timer handler.

```
function aso:onPut(value)
    self.timeout = dht.getSystemTime() + 60*MINUTES
    return self
end
function aso:onTimer()
    if (dht.getSystemTime() > self.timeout) then
        -- delete local ASO
        dht.deleteSelf()
    end
end
function aso:onGet()
    if (dht.getSystemTime() > self.timeout) then
        -- delete local ASO
        dht.deleteSelf()
        return nil
    end
    return self
end
```

Listing 2: Timeouts

An ASO can also choose to delete itself after it has been read – providing a "limited-read value" – where each replica can be read at most once. In addition to its use for self-destructing data, limited-read values could be used in settings where objects represent tasks and are deleted once they have been claimed by worker nodes. The object then serves as a synchronizing construct between the task's producer and consumer.

Listing 3 implements limited-read values. When a `get` is performed, the node records the fact that the value has been read. It then propagates the request to every other replica by overwriting them with `nil`. Note that the object does not delete itself immediately, but rather stays around for a while and periodically attempts to delete other replicas to ensure that copies on nodes with transient connectivity issues [22] are eventually deleted. Note also that concurrent `gets` issued to different replica nodes might successfully read the value. In general, as with other distributed storage systems, consistent update of replicated values would require the use of heavy-weight consensus operations. Comet does not currently provide such primitives. ASO handlers do however provide the ability for replicas to detect and correct inconsistencies, e.g., ASOs can compare and reconcile replica contents through periodic invocations of the `onTimer` handler.

```
function aso:onGet()
    if (self.read) then return nil end
    self.read = dht.getSystemTime() + 30∗MINUTES
    dht.put(dht.getKey(), nil) −−deletes replicas
    return self
end
function aso:onTimer()
    if (self.read) then
        dht.put(dht.getKey(), nil) −−deletes replicas
        if (dht.getSystemTime() > self.read) then
            dht.deleteSelf()
        end
    end
end
```

Listing 3: Limited-Read Values

Data Subscription: An ASO can allow clients to "subscribe" so that they will be notified when the ASO receives a new value. In Listing 4, when the subscriber performs a `get`, the ASO saves the subscriber's network location (`callerNode`) and a key that will serve as the subscriber's recipient of the value (`callbackKey`). When a value `update` occurs, the ASO distributes the value to all registered subscribers – the runtime ensures that the ASO distributes these values *only* to clients who have actually performed a `get` on the ASO. In the example shown, the ASO clears its subscriber list after its `put` operations; subscribers must then re-subscribe if they're still interested. Later we will describe an implementation of a scalable publish-subscribe scheme based on this design.

Sensitive values: ASOs can implement various forms of access control policies. For instance, Listing 5 provides read access to the object's value only if the client can present a predetermined password akin to a feature already provided by some DHTs, like OpenDHT [40]. A client provides the password as an argument to the `get` request.

```
aso.pending = {}
function aso:onGet(callerNode, callbackKey)
    if(self.value) then
        return self.value
    end
    self.pending[callerNode] = callbackKey
    return nil
end
function aso:onUpdate(callerNode, value)
    self.value = value
    for callerNode,key in pairs(self.pending) do
        dht.put(key, value, {callerNode})
    end
    self.pending = {}
end
```

Listing 4: Pub-sub

There are a few issues with the code provided above, especially if it were to be extended to support password-protected updates. A malicious node could claim to store the object but simply serve as a proxy for clients' requests and thereby implement man-in-the-middle attacks. This could be solved by exposing basic encryption primitives to the ASO, like a secure hash function and/or public key cryptographic primitives. For example, instead of passing the plaintext password to the ASO, the client hashes the concatenation of the password with its IP/port, thus the ASO can verify that the request is not being forwarded by a malicious node. The ASO's security can be further strengthened by public/private key pairs, with the ASO storing the public key and clients authenticating themselves by presenting a message signed with the corresponding private key. With these enhancements, a malicious node storing a copy of the object cannot overwrite the contents of other replicas since it doesn't possess the private key.

```
function aso:onGet(caller, callerId, password)
    if (password == ''mypass1234'') then
        return ''Well kept secret''
    end
    return nil
end
```

Listing 5: Password

An application could use multiple mechanisms for controlling data access, e.g., it could use timeouts in conjunction with password-protected access. While Comet does not allow ASOs to register multiple handlers for a given storage operation, the developer can combine all of the desired mechanisms into a single handler. Though this might increase programming complexity, it allows the application developer to control how different mechanisms interact with each other and provides the basis for a predictable and deterministic execution model.

5.3 Measurements and Monitoring

DHT Measurements: ASOs provide a platform for instrumenting and measuring the DHT using the DHT nodes themselves. This enables a more detailed and comprehensive view of the DHT and helps provide accurate estimates of DHT properties such as churn, node lifetime distribution, transient inconsistencies, etc.

For instance, Listing 6 tracks the *k* closest nodes to the ASO and stores the information it learns as part of the object state. A measurement application can create objects of this type, store them at multiple locations within the DHT, and obtain snapshots of DHT membership by retrieving the objects' contents using `get` operations.

```
aso.neighbors = {}
function aso:handleLookup(nodes)
    self.neighbors[dht.getSystemTime()] = nodes
end
function aso:onTimer()
    dht.lookup(dht.getKey(), self.handleLookup)
end
```

Listing 6: Lifetime

While this measurement could be performed by nodes that are not part of the DHT (as in earlier work [20, 50]), measurements from within the DHT can provide more accurate data. For example, the lifetime measurement could be carried out by a client that interactively crawls the routing tables of the DHT nodes and then uses heartbeat messages to monitor the uptimes of the nodes it learns about. This approach could provide faulty data, however, if the DHT contains firewalled nodes that do not receive or respond to such heartbeat messages.[2] On the other hand, firewalled nodes still communicate with neighbors, for example to replicate values. Therefore, measurements performed from ASOs *within* the DHT can be more accurate, as we will demonstrate later.

Monitoring uses: An ASO can also maintain audit trails, e.g., indicating where it has been stored thus far, who has read or updated the object, etc. Such tasks are particularly useful for debugging and aid in rapid prototyping. For example, this may help a developer to learn whether a new ASO replication mechanism is operating properly. Alternately, logs can also be used for forensics. Listing 7 illustrates a monitoring application that tracks the nodes storing and accessing a value.

This specific implementation comes with a few caveats. Each replica may have a different view of the list of nodes that have stored or read the value. To address this, the experimenter needs to get the union of the lists stored in all the replicas, consolidating them as a post-processing step.

[2] In fact, about half the nodes in P2P DHTs are firewalled [23].

```
replicaIps, hostIps, accessorIps = {}
function aso:onGet(callerIp)
    table.insert(self.accessorIps, callerIp)
    return self
end
function aso:onPut(caller)
    table.insert(self.accessorIps, caller.getIP())
    table.insert(self.hostIps, dht.localNode.getIP())
    return self
end
function aso:handlePut(nodes)
    for i,v in ipairs(nodes) do
        table.insert(self.replicaIps, v.getIp())
    end
end
function aso:onTimer()
    dht.put(dht.getKey(), self, 20, self.handlePut)
end
```

Listing 7: Monitoring

5.4 Smart Rendezvous

DHTs are used for rendezvous in many distributed systems. In P2P file-sharing systems such as BitTorrent, the DHT is used as a distributed tracker either with or as a replacement for a centralized tracker. That is, peers that want to download a particular file use the DHT to identify other peers who are downloading or sharing the file. The downside with current DHT-based distributed trackers, however, is that they result in random overlay connections, as there is no mechanism to enforce more intelligent peer-matching techniques.

With Comet we can address this limitation by using ASOs to track participating nodes, as well as construct peer lists that are optimized for a requesting node. Peers could be matched in order to lower inter-node latencies [33], maximize reciprocation probability based on peer bandwidths [37], or lower ISP costs [62, 12]. We have implemented one such matching scheme that uses the nodes' network coordinates to predict inter-node latencies and provides a list of nearby peers to each joining node. We describe this in depth in Section 6.3.2.

5.5 Signed ASOs

The examples discussed so far adhere to the strict security policy we set out: ASOs cannot perform operations on arbitrary DHT keys or nodes. We now consider uses where we relax this assumption, but require that the ASO code be *signed* by the DHT administrator after manual verification of its security properties. As we will see below, this allows the DHT to deploy new functionality and services by using signed ASOs that access arbitrary DHT locations, but are safe (i.e., they do not enable DoS attacks of targeted DHT nodes).[3] We have considered signed

[3] In some cases, the safety of the ASO code could presumably be verified automatically, e.g., by using sophisticated compile-time analysis;

ASOs in particular as a mechanism that a DHT's developer or administrator could use for testing and evaluation of new features, before they are added to the main-line DHT code.

Recursive Get: Vuze and many other DHTs support iterative routing for key lookups. In this approach, the node performing the lookup is involved in every step of the routing operation, i.e., it identifies the target node by repeatedly querying DHT nodes to find other nodes that are closer to the target key. An alternative is to perform recursive routing, where intermediate nodes on the route pass the lookup directly to nodes that are closer to the key. Iterative lookup provides greater control to the node performing the lookup (e.g., it can control lookup parallelism), but it comes at the cost of increased latency. If both forms of lookup are available, an application would use recursive lookups by default, but fall back on iterative lookups after persistent failures [15].

With signed ASOs it is possible to implement recursive lookups even though the underlying DHT supports iterative lookup by default (as is the case with Chord, Kademlia, and Vuze). The node initiating the lookup creates a `query ASO`, which contains a reference to itself, and a local callback ID where it would like to receive the answer. When the signed ASO is created its `onPut` handler is invoked; the handler queries the local routing table to find a live node that is closest to the target key, stores a copy of the signed ASO on this node, and deletes itself from the current node. This process is repeated until one of the nodes storing the target is reached, and the `onUpdate` handler of the target ASO sends the object's value back to the original node, which initiated the request.

Caching and Hierarchical Publish-Subscribe: This idea can be extended to accomplish both caching and hierarchical publish-subscribe data delivery. For caching, the `onUpdate` handler can be modified to communicate the object not only to the requesting node but also to the intermediate node that conveyed the request. The number of intermediate nodes to which the object is replicated can be determined by gathering and analyzing statistics on object popularity (also accomplished using simple handler code), so that only popular objects are replicated at multiple nodes (as in Beehive [39]). To implement hierarchical publish-subscribe, intermediate nodes propagate a subscription event to the next node in the lookup process only if they haven't done so before and maintain state for subsequent queries routed to them. When a value is published, it is propagated through a dissemination tree so that the communication load is distributed across all intermediate nodes (as in Scribe and Bayeux [45, 64]).

studying this is part of future work.

5.6 Summary

This section described a set of example storage objects that we have implemented using Comet. Through these examples, it should be clear that with very small extensions (on the order of a few lines or a few tens of lines of code), a Comet application can create a wide range of powerful storage object behaviors that would be impossible in existing distributed storage systems or DHTs.

6 Evaluation

We deployed Comet on approximately 200 PlanetLab hosts and evaluated our design in three steps. First, we characterize the resource utilization of the various applications that we developed. Second, we measured microbenchmarks to understand the overheads associated with active storage objects. Lastly we report on our experiences with prototyping applications using Comet.

6.1 Application Characteristics

Table 2 shows resource consumption requirements for our Comet applications. The *Max Instructions* column gives the number of dynamic Lua instructions required to execute the most expensive handler, while *Execution Time* gives the execution time for that handler. Where this value is data sensitive, we provide an estimate based on the expected maximum value. *Code Size* shows the size of each ASO with the minimum amount of data and *Max Size* is the maximum size to which the ASOs might grow for that application. From the table we see that most ASOs execute fewer than 100 Lua instructions and are smaller than 1KB in size.

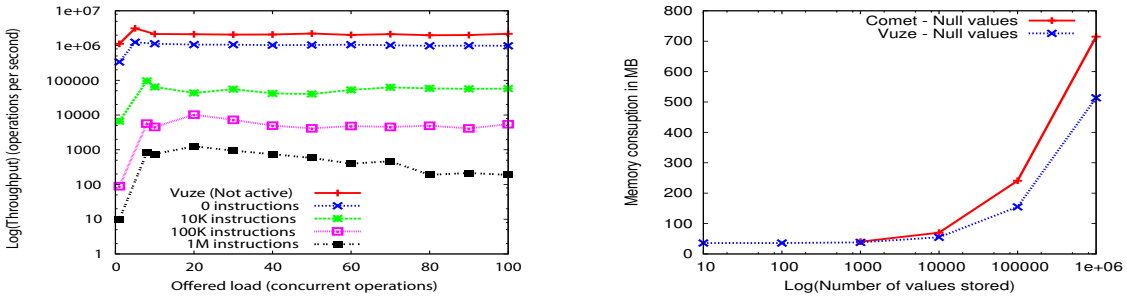
Application	Max Instructions	Execution Time	Code Size	Max Size
Replication	< 10	$4\mu s$	$0.223K$	$< 1K$
Smart Replication	< 100	$6\mu s$	$0.444K$	$< 1K$
Timeouts	≈ 10	$4\mu s$	$0.434K$	$< 1K$
Limited-Read Value	≈ 10	$4\mu s$	$0.553K$	$< 1K$
Sensitive Value	< 10	$4\mu s$	$0.230K$	$< 1K$
Pub Sub	$10,000$s	$54\mu s$	$0.498K$	$100K$
Hierarchical Pub Sub	100s	$6\mu s$	$0.673K$	$1K$
Lifetime (External)	100s	$6\mu s$	$1K$	$6K$/hr
Lifetime (Internal)	< 100	$6\mu s$	$1.776K$	$\approx 3K$
Monitoring	≈ 10	$4\mu s$	$0.971K$	$3K$/hr
Smart Rendezvous	$1,000$s	$14\mu s$	$1.107K$	$10K$
Recursive Get	≈ 50	$6\mu s$	$0.714K$	$\approx 1K$

Table 2: Expected Application Resource Consumption

6.2 Performance and Overheads

We report on simple microbenchmark measurements to compare the CPU and memory costs of Vuze and Comet. These experiments were run on an quad-core machine with Xeon processors clocked at 2.67GHz.

Single-Node Throughput. In this experiment, concurrent `get` operations are performed on many values stored in the target node. We measure the throughput of `get`



(a) Single-Node Throughput.

(b) Memory Footprint.

Figure 2: **Microbenchmarks.**

requests that return successfully using a closed feedback loop. All operations are issued locally on the node, so that network latency does not affect throughput.

Figure 2(a) compares the throughput of objects with different ASO execution costs, expressed as the number of Lua bytecode instructions executed per handler. Both Comet and Vuze experience peak throughput when the number of concurrent operations is equal to the number of cores (eight). ASOs with zero instructions per handler are functionally equivalent to Vuze values as they simply return themselves. The peak throughput of Comet ASOs is about 60% smaller than the peak throughput of Vuze (1.4M operations per second as opposed to 3.5M operations per second). This shows the cost of the Comet/Lua execution environment. Previous measurements [49] show that the typical DHT load on Vuze clients in the wild is at most a few hundred operations per second, which makes the additional Comet overhead relatively insignificant in this context. As we increase the computational complexity of the average ASO (1K to 1M instructions per handler), the throughput decreases, but still remains well above the maximum current Vuze workload.

Operation Latency. At the 90th percentile, with maximum throughput (8 concurrent operations in our experiments), a request involving 100 Lua instructions has a latency of about 300 microseconds. For handlers with 1M instructions (two orders of magnitude more than our most compute-intensive handlers), it is 13 milliseconds. The latency for a Vuze DHT lookup is on the order of seconds, therefore the latency imposed by even extremely computationally intensive ASOs is not significant.

Memory Footprint. In this experiment, we store increasing numbers of values in the nodes. For the Vuze nodes, the string "hello world" is stored at different keys, while for Comet nodes we store an equivalent Lua ASO which returns the same string upon a `get` request. Figure 2(b) compares the memory footprint of the Vuze and Comet nodes as we increase the number of stored objects. Again using the median number of values stored per Vuze node (around 400), the difference in memory consumption at this level is negligible (about 36MB for both Comet and Vuze). Long lived DHT nodes can store 10,000s of values, and the highest observed is around 30,000 values [49]. In these rare cases, our overhead relative to Vuze is about 27%, but even then the total memory footprint is still reasonable.

We next consider a workload where Comet object sizes are exponentially distributed with an average size of 10KB. In this case, a node with 500MB can store on average 50,000 values. If we assume an order of magnitude more values per node than in Vuze (4,000 instead of 400), and an order of magnitude larger values (10KB instead of 1KB limit imposed by Vuze), the median node would consume about 80MB (40MB of startup memory costs and another 40MB for the ASOs) in memory.[4]

6.3 Application Experience

We now report on our experiences in prototyping and deploying some of the applications described in Section 5.

6.3.1 Measuring Node Lifetimes

We revisit the experiment performed by Falkner et al. [20] to measure the lifetimes of nodes in the Vuze DHT. This experiment was done by performing random get operations from several Vuze clients in order to gather approximately 300K IPs participating in the DHT. The collection of nodes was then pinged every 2.5 minutes to check for liveness. The authors observed that nearly half the nodes were immediately unavailable after first being detected. One weakness of the methodology employed is that the clients could not differentiate nodes that are unreachable because of NATs from those that have left the DHT. Using measurement nodes that have active communication channels with NATed DHT nodes would help minimize

[4] Vuze and Comet consume about 40MB without storing any values.

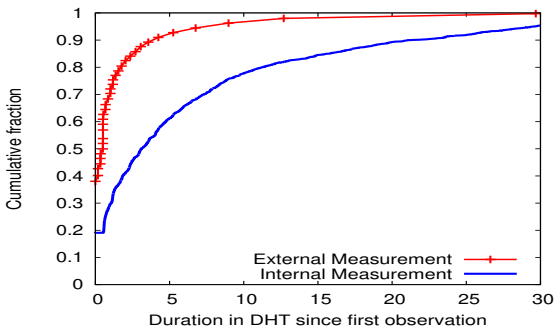


Figure 3: **Node Lifetimes in Vuze.**

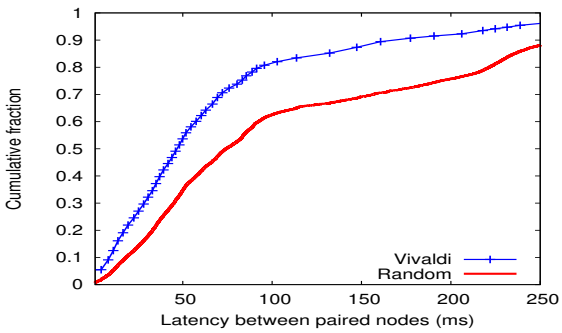


Figure 4: **Proximity of BitTorrent peers.**

measurement bias, but would require the measurement to be performed by nodes that are *within* the DHT.

Comet enables researchers to deploy experiments using measurement ASOs executed on nodes that are part of the DHT. To demonstrate the feasibility of this approach, we deployed Comet on 190 geographically dispersed PlanetLab nodes and integrated them into the production Vuze DHT. The measurement ASOs are stored on the Comet nodes, and they gather information about unmodified Vuze nodes that are adjacent (in the DHT) to the Comet nodes. We stored a lifetime measurement ASO (a variant of the code shown in Listing 6) at each of the Comet nodes, allowed the nodes to perform measurements for several days, and then collected and analyzed the data from these nodes.[5] Figure 3 plots the measurement data obtained from our experiments and compared to the lifetime data obtained by measurement nodes that are not integrated into the DHT (as in [20]). We observe that the measurements performed from within the DHT provide higher estimates for node lifetimes. The reason is that DHT-internal measurement nodes are able to traverse NATs in communicating with their neighbors. The difference is significant; we measured the median node lifetime as 3.1 hours, as opposed to an estimate of 0.5 hours obtained through conventional external measurements. Measurement ASOs are thus valuable tools in characterizing DHTs and provide more accurate data for tuning parameters such as replication factor, routing parallelism, etc.

6.3.2 Smart Rendezvous

In Section 5, we proposed a way to employ intelligent peer tracking for distributed P2P trackers using ASOs. We evaluate the usefulness of this application by deploying a distributed tracker built with Comet ASOs. As with traditional distributed trackers, clients participating in a P2P swarm (such as a BitTorrent download) register their participation by storing their IP addresses under the appropriate DHT key. In addition, clients also store their network coordinates (computed using Vivaldi [13]) along with their IP information. When clients contact the distributed tracker to obtain peer lists, the tracker ASO estimates the network latency between pairs of nodes using the supplied network coordinates and returns peers that are likely to be close to the requesting node. To evaluate this approach in practice, we deployed a tracker ASO on a Comet node in PlanetLab, while 190 PlanetLab nodes acted as peers in the swarm reporting their Vivaldi coordinates to the tracker and requesting good peers with which to communicate. Figure 4 depicts the effectiveness of this strategy compared to the default strategy of returning a random subset of peers to the requesting node. The graph shows a CDF of the measured latencies between peers under the two different matching schemes. The median value for the ASO-implemented Vivaldi intelligent peer matching is 47ms compared to a median of 72ms for the default scheme, a 35% latency improvement.

6.3.3 Vanish

Comet grew in part out of our experience specializing the Vuze DHT for Vanish [25], a self-destructing data system. Vanish used Vuze for key storage, however, Wolchock et al. [61] showed that the Vuze system was extremely open to a Sybil data harvesting attack that is able to scan the DHT for values. The attack worked in part because of Vuze's overly zealous replication policy – a high replication factor, coupled with a policy to replicate to new nodes immediately. In response, we set out to deploy new replication mechanisms and other anti-Sybil defenses in Vuze [24]. While these mechanisms were straightforward, deploying them required the cooperation of Vuze's designer and was an arduous and imperfect process. While many iterations would have been necessary to fully test and optimize policies, we often had only one shot to catch the two-month release cycle.[6]

[5] As Comet is not currently deployed by Vuze, the measurement ASOs are stored only on the nodes that we control. A more extensive deployment would allow us to obtain more samples quickly.

[6] It takes a week or more from release until 80% of the nodes in Vuze adopt changes. This is in addition to a typical release cycle Vuze

For the same reason we were unable to test individual changes in isolation as they had to be shipped in bundles in order to make progress in reasonable time.

We have used Comet to re-implement several of the changes that we deployed in Vuze. Those changes include the customizable replication scheme described in Section 5 (particularly a scheme that replicates only when the number of replicas falls below a threshold) and variable object lifetimes. As we showed in Section 5, both of these changes are trivial to program as Comet ASOs. Perhaps even more important, testing and re-deployment in Comet is significantly easier, as it does not require a redistribution of the entire DHT code base. Instead, new mechanisms can be deployed by overwriting the handler code for existing objects and using the updated bytecode for subsequently created objects, without requiring the involvement of the DHT administrators.[7] Had Comet existed at the time we deployed Vanish, it would have been possible to customize the DHT for the security requirements of the application from the start, and to optimize those policies to Vanish's requirements.

7 Security Analysis

The classic security goals for DHTs include resilience to attacks that: violate the system's ability to robustly store data [48], disrupt routing [48, 11], identify the participating nodes in the DHT [53, 51], and harvest copies of data stored within the DHT [61]. There are numerous well-known techniques aimed at violating these goals, including Sybil attacks [19], Eclipse attacks [47], and many others [55]. And there are also many known mechanisms for protecting against such attacks, including the use of strong identities minted by a logically centralized authority, computational puzzles and bandwidth contributions proofs [9, 16, 18, 63, 10], and architectures built upon social network structures [31, 63]. A production DHT with ASO support must consider such classic security goals, and can leverage known countermeasures for the corresponding threats. (Although, as exemplified by Vuze and other popular DHTs, a DHT for ASOs may decide that the risks associated with these threats are minimal, and hence not deploy the known defenses.)

The security concerns of DHTs with *signed* ASOs are roughly those of conventional DHTs without ASOs (since the signed ASOs can be viewed as "vetted" parts of the DHT system itself); we therefore do not consider signed ASOs further. Empowering DHTs with *unsigned* ASOs does, however, create a *new* potential attack vector not present in conventional DHTs – namely, attacks via malicious ASOs. We seek to ensure that a malicious ASO cannot: infer private information about or damage its Comet hosting node; infer information about or affect the properties of other ASOs stored within Comet; or infer private information about or affect the properties of other Comet nodes and arbitrary computers on the Internet. To place these goals in context, we stress that while an attacker could always use her own custom software to communicate with Comet in arbitrary ways, including putting to or getting from arbitrary ASO keys and communicating with the broader Internet in arbitrary ways, our goals – if attained – imply that ASOs cannot be used to amplify the attacker's resources or capabilities. For example, an attacker should not be able to create an ASO "worm" that spreads virally, mounting a DDoS attack against a victim ASO or device on the Internet.

We find that it is possible to meet these goals using three architectural features: (1) restricting system access, (2) restricting resource consumption, and (3) restricting within-Comet communication. We consider each in turn.

Restricting system access. We designed the ASO API to be highly restrictive. The API explicitly restricts an ASO's ability to infer private information about its host or to affect the host's state. The API similarly restricts an ASO's ability to interact with arbitrary devices on the Internet. For example, the API limits an ASO's IO capabilities to explicitly defined DHT operations; arbitrary disk, network, and other IO operations are prohibited. The API also prevents an ASO from introspecting its host; e.g., although we allow the ASO to learn its host's external IP, we explicitly prevent the ASO from learning its host's internal IP. Without these restrictions, an ASO could potentially read private files on the host's disk, write sensitive files, attempt to DoS an arbitrary remote node, map the network topology of internal IP networks, and so on. The Lua sandbox provides a simple mechanism for achieving this isolation. Namely, we removed the IO system call interface and exposed one containing only the restricted DHT operations instead.

Despite these restrictions, it may be possible for an ASO to infer (minimal) information about the hosting node via side-channels. For example, the time it takes an ASO to perform a computation could leak information to the ASO about the speed of the hosting processor. At the extreme, it may be feasible to infer modest information about other applications running on the hosting node [42]. We believe that such attacks are low risk in the Comet environment and do not consider them here.

Restricting resource consumption. Comet also significantly limits an ASO's ability to consume resources on its hosting node. Our prototype limits both the memory and CPU consumption of ASOs.

Memory. The Comet active runtime keeps a running

employs, which spans about a month.

[7] In general, updating the handler code for existing objects would require the application to keep track of its ASOs. In the case of applications such as Vanish, where objects are transient and have timeouts in the order of a few hours, we can also let existing objects just expire without explicitly updating them.

sum of the memory footprint of an ASO. Hard limits can be set on the total memory consumption of an object; ASOs which exceed this limit are evicted. Our current prototype limits ASOs to 100kB.

CPU. The Comet runtime similarly keeps a running count of bytecode operations performed. We envision multiple policies for constraining CPU use. The naive policy limits each ASO to at most a limited number of instructions per handler invocation. Since not all Lua operations are equally costly, a more sophisticated policy would assign different weights to different Lua operations (e.g., more cost for a table lookup than an addition). The limit could also be enforced over a fixed duration of time (such as 30 minutes) rather than upon each handler invocation (which might occur much more frequently). Our current prototype implements the naive restriction and allows 100K instructions per handler invocation.

Comet provides support for exception handling in order to help debug faulty ASOs that exceed the system-imposed resource limits. Handlers can catch resource exhaustion exceptions and store the relevant handler state as part of the ASO. The developer can then retrieve this stored state and inspect it to determine why the handler exceeded the resource limits. Further, operations that return values, e.g., `gets`, provide the stack trace as a return value in the case of an exception. We found these features to be useful in debugging many of the applications that we prototyped using Comet.

Restricting within-Comet communications. There are two classes of communications that we must consider: communications between one ASO and another, and callback communications to a caller.

Communications between ASOs. Allowing arbitrary between-ASO communications in Comet could lead to abuse. For example, suppose a malicious ASO stored under one key copies itself to a large number of other keys slowly over time, and then simultaneously all ASOs initiate connections to a victim ASO stored under some target key. Such an attack allows an attacker to amplify her resources: the attacker invests minimal effort to seed the original malicious ASO, yet the ultimate attack DDoSes nodes hosting the target key. Comet takes a Draconian approach toward protecting against such attacks: the ASO API only allows ASOs to communicate if they are stored under the *same* key, whether co-located on the same Comet node or on another node within the DHT. Our system further rate-limits communications performed by a particular ASO. Each Comet node allots a limited number of network communications per time period for every ASO it hosts. Though we have not experimentally ascertained appropriate rate-limiting parameters, the applications we present could all work with approximately the same number of network operations as is required for a value in the current Vuze DHT - about 20 every timer interval.

8 Conclusions

This paper described Comet, an active distributed key-value store. Comet enables clients to customize a distributed storage system in application-specific ways using Comet's active storage objects. By supporting ASOs, Comet allows multiple applications with diverse requirements to share a common storage system. We implemented Comet on the Vuze DHT using a severely restricted Lua language sandbox for handler programming. Our measurements and experience demonstrate that a broad range of behaviors and customizations are possible in a safe, but active, storage environment.

9 Acknowledgements

This work was supported in part by the National Science Foundation under grants NSF-0627367, NSF-0614975, NSF-0619836, NSF-0722004, and NSF-0963754, by the Google Fellowship in Cloud Computing, and by the Wissner-Slivka Chair. We thank Paul Gardner for his support on Vuze, and David Wetherall and our shepherd Wilson Hsieh for their helpful feedback on the paper.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evalaution. In *Proc. of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[2] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of OSDI*, 2002.

[3] Amazon S3. `http://aws.amazon.com/s3/`.

[4] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet impasse through virtualization. *IEEE Computer*, 38(4), April 2005.

[5] Apache Cassandra. `http://cassandra.apache.org/`.

[6] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensible, safety and performance in the SPIN operating system. In *Proc. of the 15th ACM Symp. on Operating systems Principles*, December 1995.

[7] B. N. Bershad and C. B. Pinkerton. Watchdogs – extending the UNIX file system. *Computer Systems*, 1(2), 1988.

[8] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proc. of SIGCOMM*, 2004.

[9] N. Borisov. Computational puzzles as Sybil defenses. In *Proc. of the Intl. Conference on Peer-to-Peer Computing*, 2006.

[10] N. Borisov. Computational puzzles as Sybil defenses. In *Proc. of the Intl. Conference on Peer-to-Peer Computing*, 2006.

[11] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.*, 2002.

[12] D. R. Choffnes and F. E. Bustamante. Taming the Torrent: A practical approach to reducing cross-ISP traffic in P2P systems. In *Proc. of SIGCOMM*, 2008.

[13] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: a decentralized network coordinate system. In *Proc. of SIGCOMM*, 2004.

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of SOSP*, 2001.

[15] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a dht for low latency and high throughput. In *NSDI*, 2004.

[16] G. Danezis, C. Lesniewski-Laas, M. F. Kaashoek, and R. J. Anderson. Sybil-resistant DHT routing. In *ESORICS*, 2005.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of SOSP*, 2007.

[18] J. Dinger and H. Hartenstein. Defending the Sybil Attack in P2P Networks: Taxonomy, Challenges, and a Proposal for Self-Registration. In *Intl. Conf. on Availability, Reliability and Security*, 2006.

[19] J. R. Douceur. The Sybil attack. In *Proc. of IPTPS*, 2002.

[20] J. Falkner, M. Piatek, J. P. John, A. Krishnamurthy, and T. Anderson. Profiling a million user DHT. In *Proc. of IMC*, 2007.

[21] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with coral. In *NSDI*, pages 239–252, 2004.

[22] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. In *WORLDS'05*, pages 10–10, Berkeley, CA, USA, 2005. USENIX Association.

[23] P. Gardner. personal communication, 2009.

[24] R. Geambasu, T. Kohno, A. Krishnamurthy, A. Levy, H. M. Levy, P. Gardner, and V. Mascaritolo. Cascade: A compositional approach to self-destructing data. In Preparation, 2010.

[25] R. Geambasu, T. Kohno, A. Levy, and H. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proc. of the USENIX Security Symposium*, August 2009.

[26] K. Hildrum and J. Kubiatowicz. Asymptotically Efficient Approaches to Fault-Tolerance in Peer-to-peer Networks. In *Proc. of International Symposium on Distributed Computing*, 2004.

[27] T. Isdal, M. Piatek, A. Krishnamurthy, and T. Anderson. Privacy-preserving P2P data sharing with OneSwarm. In *Proc. of SIGCOMM*, 2010.

[28] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, , and K. Mackenzie. Application performance and flexibility in exokernel systems. In *Proc. of SOSP*, 1997.

[29] K. Keetong, D. Patterson, and J. Hellerstein. A case for intelligent disks (IDISKs). *ACM SIGMOD Record*, 27(3), August 1998.

[30] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, December 1999.

[31] C. Lesniewski-Lass and M. F. Kaashoek. Whanaungatanga: Sybil-proof distributed hash table. In *Proc. of NSDI*, 2010.

[32] N. A. Lynch, D. Malkhi, and D. Ratajczak. Atomic Data Access in Distributed Hash Tables. In *Proc. of IPTPS*, 2001.

[33] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An Information Plane for Distributed Services. In *OSDI*, 2006.

[34] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: A fault-tolerant algorithm for atomic mutable DHT data. Technical Report MIT-LCS-TR-993, MIT, June 2005.

[35] Mysql Database Triggers. `http://dev.mysql.com/doc/refman/5.0/en/triggers.html`.

[36] L. Peterson, A. Bavier, M. Fiuczynski, and S. Muir. Experiences implementing PlanetLab. In *Proc. of OSDI*, 2006.

[37] M. Piatek, T. Isdal, A. Krishnamurthy, and T. Anderson. Do incentives build robustness in BitTorrent? In *NSDI*, 2007.

[38] Project Voldemort. `http://project-voldemort.com/`.

[39] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proc. of NSDI*, Berkeley, CA, USA, 2004. USENIX Association.

[40] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proc. of SIGCOMM*, 2005.

[41] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proc. of 24th International Conference on Very Large Databases*, August 1998.

[42] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. of CCS*, 2009.

[43] W. C. F. Roberto Ierusalimschy, Luiz Henrique de Figueiredo. Lua - an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1999.

[44] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of SOSP*, 2001.

[45] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Proc. of the Third International COST264 Workshop on Networked Group Communication*, 2001.

[46] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *OSDI*, 1996.

[47] A. Singh, T.-W. Ngan, P. Druschel, , and D. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *INFOCOM*, 2006.

[48] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Proc. of IPTPS*, 2002.

[49] M. Steiner and E. W. Biersack. Crawling AZUREUS. Technical report, Institut Eurecom, Networking and Security Department, 2008.

[50] M. Steiner, E. W. Biersack, and T. Ennajjary. Actively monitoring peers in KAD. In *Proc. of IPTPS*, 2007.

[51] M. Steiner, T. En-Najjary, and E. W. Biersack. A Global View of KAD. In *Proc. of IMC*, 2007.

[52] I. Stoica, D. Adkins, S. Zhuang, S. S. nker, and S. Surana. Internet indirection infrastructure. In *Proc. of SIGCOMM*, 2002.

[53] D. Stutzbach and R. Rejaie. Understanding Churn in Peer-to-Peer Networks. In *Proc. of IMC*, 2006.

[54] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *ACM SIGCOMM Computer Communications Review*, April 1996.

[55] G. Urdaneta, G. Pierre, and M. V. Steen. A Survey of DHT Security Techniques (to appear). *ACM Computing Survey*, 2010.

[56] uTorrent. `http://www.utorrent.com`.

[57] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. of ISCA*, 1992.

[58] Vuze, Inc. `http://www.vuze.com`.

[59] P. Wang, I. Osipkov, N. Hopper, and Y. Kim. Myrmic: Secure and Robust DHT Routing. Technical report, University of Minnesota, 2007.

[60] D. Wetherall. Active network vision and reality: Lessons from a capsule-based system. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, December 1999.

[61] S. Wolchok, O. S. Hofmann, E. W. Felten, J. A. Halderman, C. J. Rossbach, B. Waters, and E. Witchel. Defeating Vanish with low-cost Sybil attacks against large DHTs. In *Proc. of NDSS*, 2010.

[62] H. Xie, R. Yang, A. Krishnamurthy, Y. Liu, and A. Silberschatz. P4P: Provider portal for P2P applications. In *Proc. of SIGCOMM*, 2008.

[63] H. Yu, M. Kaminsky, P. B. Gibbons, and A. D. Flaxman. SybilGuard: defending against sybil attacks via social networks. *Proc. of SIGCOMM*, 2006.

[64] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. of NOSSDAV*, 2001.

SPORC: Group Collaboration using Untrusted Cloud Resources

Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten
Princeton University

Abstract

Cloud-based services are an attractive deployment model for user-facing applications like word processing and calendaring. Unlike desktop applications, cloud services allow multiple users to edit shared state concurrently and in real-time, while being scalable, highly available, and globally accessible. Unfortunately, these benefits come at the cost of fully trusting cloud providers with potentially sensitive and important data.

To overcome this strict tradeoff, we present SPORC, a generic framework for building a wide variety of collaborative applications with *untrusted* servers. In SPORC, a server observes only encrypted data and cannot deviate from correct execution without being detected. SPORC allows concurrent, low-latency editing of shared state, permits disconnected operation, and supports dynamic access control even in the presence of concurrency. We demonstrate SPORC's flexibility through two prototype applications: a causally-consistent key-value store and a browser-based collaborative text editor.

Conceptually, SPORC illustrates the complementary benefits of *operational transformation* (OT) and *fork* consistency*. The former allows SPORC clients to execute concurrent operations without locking and to resolve any resulting conflicts automatically. The latter prevents a misbehaving server from equivocating about the order of operations unless it is willing to *fork* clients into disjoint sets. Notably, unlike previous systems, SPORC can automatically recover from such malicious forks by leveraging OT's conflict resolution mechanism.

1 Introduction

An emerging class of cloud-based collaborative services, such as online document processing and calendaring, provides users with anywhere-available, real-time, and concurrent access to shared state. Their deployments on managed cloud platforms enjoy global accessibility, high availability, fault tolerance, and elastic resource allocation and scaling. Yet these benefits have come at the cost of having a fully trusted server, creating a risk of privacy problems due to server-side information leaks. The history of such services is one rife with unplanned data disclosures and malicious break-ins [24]. Indeed, the very centralization of information makes cloud providers high value targets for attack. Further, the behavior of service providers themselves is a source of users' privacy angst, as privacy policies may be weakened due to market expediencies. Finally, cloud providers face pressure from government agencies world-wide to release information on demand [15].

This paper challenges the belief that applications must sacrifice strong security and privacy to enjoy the benefits of cloud deployment. We present a system, SPORC, that offers managed cloud-based deployment for group collaboration services, yet does require users to trust the cloud provider to maintain data privacy or even to operate correctly. SPORC's cloud servers see only encrypted data, and clients will detect any deviation from correct operation (*e.g.*, adding, modifying, dropping, or reordering operations) and will recover from the error. Much like SUNDR [24], SPORC bases its security and privacy guarantees on the security of users' cryptographic keys, and not on the cloud provider's good intentions nor on some threshold-like protocol between servers [9] that is susceptible to administrative or software attacks.

SPORC provides a *generic* collaboration service in which users can create a document, modify its access control list, edit it concurrently, experience fully automated merging of updates, and even perform these operations while disconnected. The SPORC framework supports a broad range of collaborative applications. Data updates are encrypted before being sent to a cloud-hosted server. The server assigns a total order to all operations and redistributes the ordered updates to clients. If a malicious server drops or reorders updates, the SPORC clients can detect the server's misbehavior, switch to a new server, restore a consistent state, and continue. The same mechanism that allows SPORC to merge correct concurrent operations also enables it to transparently recover from attacks that fork clients' views.

From a conceptual distributed systems perspective, SPORC demonstrates the benefit of combining *operational transformation* [11] and *fork* consistency* protocols [23]. Operational transformation (OT) defines a framework for executing lock-free concurrent operations that both preserves causal consistency and converges to a common shared state. It does so by transforming operations so they can be applied commutatively by different clients, resulting in the same final state. While OT originated with decentralized applications using pairwise reconciliation [11, 18], recent systems like Google Wave [44] have used OT with a trusted central server that orders and transforms clients' operations. Fork* consistency, on the

other hand, was introduced as a consistency model for interacting with an untrusted server: If the server causes the views of two clients to diverge, the clients must either never see each others' subsequent updates or else identify the server as faulty.

Recovering from a malicious fork is similar to reconciling concurrent operations in the OT framework. Upon detecting a fork, SPORC clients use OT mechanisms to replay and transform forked operations, restoring a consistent state. Previous applications of fork* consistency [23] could only detect forks, but not resolve them.

This paper makes the following contributions:

- §2 We identify and explore the conceptual connection between operational transformation protocols and the fork* consistency model, and use this connection to motivate SPORC's design.
- §3 We describe SPORC's framework and protocols for real-time collaboration. SPORC provides security and privacy against both an untrusted server that mediates communication and other clients that lack access control permissions.
- §4 We demonstrate how to support dynamic access control, which is challenging because SPORC supports concurrent operations and offline editing.
- §5 We describe how clients can detect and recover from maliciously-instigated forks. We also present a checkpoint mechanism that reduces saved client state and minimizes the join overhead for new clients.
- §6 We illustrate the extensibility of SPORC's pluggable data model by building both a key-value store and a browser-based collaborative text editor. We implement these services as both stand-alone applications and web services; the latter run in a browser, execute in JavaScript (compiled from Java via GWT [12]), and require no prior installation.

We evaluate SPORC's performance in Section 7 before discussing related work and concluding.

2 System Model

The purpose of SPORC is to allow a group of users who trust each other to collaboratively edit some shared state, which we call the *document*, with the help of an untrusted server. SPORC is comprised of a set of client devices that modify the document on behalf of particular users, and a potentially-malicious server whose main role is to impose a global order on those modifications. The server receives updates from individual clients, orders them, and then broadcasts them to the other clients. Access to the document is limited to a set of authorized users, but each user may be logged into arbitrarily many clients simultaneously (*e.g.*, her desktop, laptop, and mobile phone). Each client, even if it is controlled by the same user as another client, has its own local view of the document that must be synchronized with all other clients.

2.1 Goals

We designed SPORC with the following goals in mind:

Flexible framework for a broad class of collaborative services. Because SPORC uses an untrusted server which does not see application-level content, the server is generic and can handle a broad class of applications. On the client side, SPORC provides a library suitable for use by a range of desktop and web-based applications.

Propagate modifications quickly. When a client is connected to the network, its changes to the shared state should propagate quickly to all other clients so that clients' views are nearly identical. This property makes SPORC suitable for building collaborative applications requiring nearly real-time updates, such as collaborative text editing and instant messaging.

Tolerate slow or disconnected networks. To allow clients to edit the document while offline or while experiencing high network latency, clients in SPORC update the document *optimistically*. Every time a client generates a modification, the client applies it immediately to its local state, and only later sends it to the server for redistribution. As a result, clients' local views of the document will invariably diverge, and SPORC must be able to resolve these divergences automatically.

Keep data confidential from the server and unauthorized users. Since the server is untrusted, document updates must be encrypted before being sent to the server. For efficiency, the system should use symmetric-key encryption. SPORC must provide a way to distribute this symmetric key to every client of authorized users. When a document's access control list changes, SPORC must ensure that newly added users can decrypt the entire document, and that removed users cannot decrypt any updates subsequent to their expulsion.

Detect a misbehaving server. Even without access to document plaintext, a malicious server could still do significant damage by deviating from its assigned role. It could attempt to add, drop, alter, or delay clients' (encrypted) updates, or it could show different clients inconsistent views of the document. SPORC must give clients a means to quickly detect these kinds of misbehavior.

Recover from malicious server behavior. If clients detect that the server is misbehaving, clients should be able to failover to a new server and resume execution. Since a malicious server could cause clients to have inconsistent local state, SPORC must provide a mechanism for automatically resolving these inconsistencies.

To achieve these goals, SPORC builds on two conceptual frameworks: *operational transformation* and *fork* consistency*.

2.2 Operational Transformation

Operational Transformation (OT) [11] provides a general model for synchronizing shared state, while allowing each client to apply local updates optimistically. In OT, the application defines a set of *operations* from which all modifications to the document are constructed. When clients generate new operations, they apply them locally before sending them to others. To deal with the conflicts that these optimistic updates inevitably incur, each client *transforms* the operations it receives from others before applying them to its local state. If all clients transform incoming operations appropriately, OT guarantees that they will eventually converge to a consistent state.

Central to OT is an application-specific *transformation function* $T(\cdot)$ that allows two clients whose states have diverged by a single pair of conflicting operations to return to a consistent, reasonable state. $T(op_1, op_2)$ takes two conflicting operations as input and returns a pair of transformed operations (op_1', op_2'), such that if the party that initially did op_1 now applies op_2', and the party that did op_2 now applies op_1', the conflict will be resolved.

To use the example from Nichols *et al.* [30], suppose Alice and Bob both begin with the same local state "ABCDE", and then Alice applies $op_1 =$ 'del 4' locally to get "ABCE", while Bob performs $op_2 =$ 'del 2' to get "ACDE". If Alice and Bob exchanged operations and executed each others' naively, then they would end up in inconsistent states (Alice would get "ACE" and Bob "ACD"). To avoid this problem, the application supplies the following transformation function that adjusts the offsets of concurrent delete operations:

$$T(\text{del } x, \text{del } y) = \begin{cases} (\text{del } x-1, \text{del } y) & \text{if} \quad x > y \\ (\text{del } x, \text{del } y-1) & \text{if} \quad x < y \\ (\text{no-op}, \text{no-op}) & \text{if} \quad x = y \end{cases}$$

Thus, after computing $T(op_1, op_2)$, Alice will apply $op_2' =$'del 2' as before but Bob will apply $op_1' =$ 'del 3', leaving both in the consistent state "ACE".

Given this pair-wise transformation function, clients that diverge in arbitrarily many operations can return to a consistent state by applying the transformation function repeatedly. For example, suppose that Alice has optimistically applied op_1 and op_2 to her local state, but has yet to send them to other clients. If she receives a new operation op_{new}, Alice must transform it with respect to both op_1 and op_2: She first computes $(op_{new}', op_1') \leftarrow T(op_{new}, op_1)$, and then $(op_{new}'', op_2') \leftarrow T(op_{new}', op_2)$. This process yields op_{new}'', an operation that Alice has "transformed past" her two local operations and can now apply to her local state.

Throughout this paper, we use the notation $op' \leftarrow T(op, \langle op_1, \ldots, op_n \rangle)$ to denote transforming op past a sequence of operations $\langle op_1, \ldots, op_n \rangle$ by iteratively applying the transformation function.[1] Similarly, we define $\langle op_1', \ldots, op_n' \rangle \leftarrow T(\langle op_1, \ldots, op_n \rangle, op)$ to represent transforming a sequence of operations past a single operation.

Operational transformation can be applied in a wide variety of settings, as operations, and the transforms on them, can be tailored to each application's requirements. For a collaborative text editor, operations may contain inserts and deletes of character ranges at specific cursor offsets, while for a causally-consistent key-value store, operations may contain lists of keys to update or remove. In fact, we have implemented both such systems on top of SPORC, which we describe further in Section 6.

For many applications, with a carefully-chosen transformation function, OT is able to automatically return divergent clients to a state that is not only consistent, but semantically reasonable as well. But for some applications, such as source-code version control, semantic conflicts must be resolved manually. OT can support such applications through the choice of a transformation function that does not try to resolve the conflict, but instead inserts an explicit conflict marker into the history of operations. A human can later examine the marker and resolve the conflict by issuing new writes. These write operations will supercede the conflicting operations, provided that the system preserves the global order of committed operations and the partial order of each client's operations. Section 3 describes how SPORC provides these properties.

While OT was originally proposed for decentralized n-way synchronization between clients, many prominent OT implementations are server-centric, including Jupiter [30] and Google Wave [44]. They rely on the server to resolve conflicts and to maintain consistency, and are architecturally better suited for web services. On the flip side, a misbehaving server can compromise the confidentiality, integrity, and consistency of the shared state.

Later, we describe how SPORC adapts these server-based OT architectures to provide security against a misbehaving server. At a high level, SPORC has each client *simulate* the transformations that would have been applied by a trusted OT server, using the server only for ordering. But we still need to protect against inconsistent orderings, for which we leverage fork* consistency techniques [23].

2.3 Fork* Consistency

To prevent a malicious server from forging or modifying clients' operations, clients in SPORC digitally sign all their operations with their user's private key. This is not sufficient for correctness, however: a misbehaving server could still equivocate and present different clients with divergent views of the history of operations.

[1] Strictly speaking, T always returns a pair of operations. For simplicity, however, we sometimes write T as returning a single operation, especially when the other is unchanged, as in our "delete char" example.

To defend against server equivocation, SPORC clients enforce *fork* consistency* [23].[2] In fork*-consistent systems, clients share information about their individual views of the history by embedding it in every operation they send. As a result, if clients to whom the server has equivocated ever communicate, they will discover the server's misbehavior. The server can still divide its clients into disjoint groups and only tell each client about operations by others in its group. But, once the server has *forked* two groups in this way, it cannot tell a member of one group about an operation submitted by another group's members without risking detection.

As in BFT2F [23], each SPORC client enforces fork* consistency by maintaining a *hash chain* over its view of the committed history. In this context, a hash chain is a method of incrementally computing the hash of a list of elements. More specifically, if $op_1, \ldots, op_n$ are the operations in the history, h_0 is a constant initial value, and h_i is the value of the hash chain over the history up to op_i, then $h_i = H(h_{i-1}||H(op_i))$, where $H(\cdot)$ is a cryptographic hash function and $||$ denotes concatenation. When a client with history up to op_n submits a new operation, it includes h_n in its message. On receiving the operation, another client can check whether the included h_n matches its own hash chain computation over its local history up to op_n. If they do not match, the client knows that the server has equivocated.

2.4 The Benefits of Having a Server

SPORC uses a central untrusted server, but the server's sole purpose is to order and store client-generated operations. This limited role may lead one to ask whether the server should be removed, leading to a completely peer-to-peer design. Indeed, many group collaboration systems, such as Bayou [43] and Network Text Editor [17], employ decentralized architectures. Decentralized designs are a poor fit, however, for applications in which a user needs a timely notification that her operation has been committed and will not be overridden by another's (not yet received) operation. For example, to schedule a meeting room, an online user should be able to quickly determine whether her reservation succeeded, without worrying if an offline client's request will override hers. Yet this is difficult to achieve without waiting to hear from all (or at least a quorum of) other clients, which poses a problem when clients are regularly offline. In reaction, Bayou delegates commits to a (statically) designated, trusted "primary" peer, which is little different from having a server.

SPORC, on the other hand, only requires an *untrusted* server for globally ordering operations. Thus, it can leverage the benefits of a cloud deployment—high availability and global accessibility—to achieve timely commits. We show in Section 4.2 how SPORC's centralized server also helps support dynamic access control and key rotation, even in the face of concurrent membership changes.

2.5 Deployment and Threat Model

Deployment Assumptions. While most of the paper discusses the SPORC protocol in terms of a single server and a single document, we assume that a cloud-based SPORC deployment would manage large numbers of users and documents by replicating functionality and partitioning state over many servers. Each document in SPORC can be managed independently, leading naturally to the shared-nothing architectures [36] already common to scalable cloud services.

For a client to recover from a misbehaving server, we assume there exists some alternative (untrusted) server to switch to after a client detects faulty behavior. These backup servers may belong to the same or different administrative domains as the original, depending upon the type of faults that a SPORC deployment expects to encounter.

Note that even if malicious (Byzantine) behavior among cloud servers is not a primary concern, this strong threat model also covers weaker non-crash failures related to server misconfiguration, Heisenbugs, or "split-brain" partitioned behavior. In all cases, failover and recovery is client driven. Crash failures, unlike Byzantine failures, would not result in forks and could be handled by traditional fault-tolerance techniques (*e.g.*, primary/backup replication) already employed in cloud services.

Threat Model. SPORC makes the following security assumptions:

Server: The server is potentially malicious, and a misbehaving server may be able to prevent progress, but it must not be able to corrupt the clients' shared state. A server may fork clients' states, but only within the confines of the fork* consistency model. If clients are able to communicate either in-band or out-of-band, server equivocation will be detected promptly by at least one client.

The server may be able to learn which users and clients are sharing a document, but it must not learn what is in the document or even the contents of the individual operations that the clients submit. Since the server has access to the size and timing of clients' operations, it may be able to glean some information about the document via traffic analysis. Traffic analysis is made more difficult by the fact that encrypted operations do not even reveal which portions of the shared state they modify. Nevertheless, the complete mitigation of traffic analysis is beyond the scope

[2] Fork* consistency is a weaker variant of an earlier model called *fork consistency* [27]. They differ in that under fork consistency, a pair of clients only needs to exchange one message to detect server equivocation, whereas under fork* consistency, they may need to exchange two. For OT systems like ours, this distinction makes little difference because clients constantly exchange small messages. On the other hand, fork* consistency permits a one-round protocol to submit operations, rather than two. Beyond efficiency, this also ensures that a crashed client cannot prevent the system from making progress.

of this work, but it would likely involve padding the length of operations and introducing cover traffic.

To attack availability, the server may arbitrarily erase or refuse to return any of the encrypted data that it stores. To mitigate this threat, the encrypted data could be replicated on servers in other administrative domains. Moreover, each client could replicate its own local state on cloud servers other than the main SPORC server. Notably, SPORC cannot guarantee recovery from every possible fork, unless every client stores every operation that it has seen either locally or remotely.

Clients: If a client is logged in as a particular user, that client is trusted to exercise the privileges granted to that user (*e.g.*, to see the state, modify it, or modify access privileges). Otherwise, clients are untrusted, and they should not be able to see the document, or to modify the document or its access control list, even if they collude with each other or with the server.

User authentication and keys: We assume that each user has a secure public/private key pair, and that clients have a secure way to verify the public key of other users.

Application code: We assume the presence of a code authentication infrastructure that can verify that the application code run by clients is genuine. This mechanism might rely on code signing or on HTTPS connections to a trusted server (different from the untrusted server used as part of SPORC's protocols).

3 System Design

This section describes SPORC's design in more detail, including its synchronization mechanisms and the measures that clients implement to detect a malicious server that may modify, reorder, duplicate, or drop operations. This section assumes that the set of users and clients editing a given document is fixed; we consider dynamic membership in Section 4.

3.1 System Overview

The parties and stages involved with SPORC operations are shown in Figure 1. At a high level, the local state of a SPORC application is synchronized between multiple clients, using a server to collect updates from clients, order them, then redistribute the client updates to others. There are four types of state in the system.

(1) The ***local state*** is a compact representation of the client's current view of the document (*e.g.*, the most recent version of a collaborative-edited text).

(2) The ***encrypted history*** is the set of operations stored at and ordered by the server. The payloads of operations that change the contents of the document are encrypted to preserve confidentiality. The server orders the operations oblivious to their payloads but aware of the previous operations on which they causally depend.

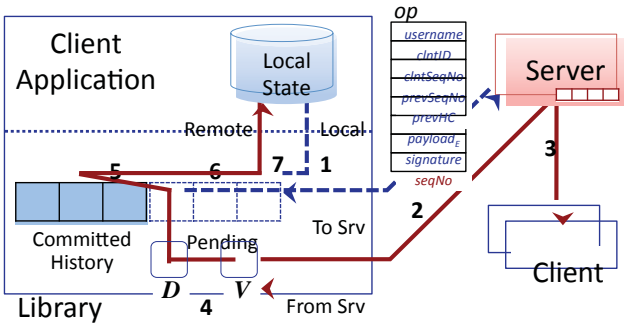


Figure 1: **SPORC architecture and synchronization steps**

(3) The ***committed history*** is the official set of (plaintext) operations shared among all clients, as ordered by the server. Clients derive this committed history from the server's encrypted history by transforming operations' payloads to reflect any changes that the server's ordering might have caused.

(4) A client's ***pending queue*** is an ordered list of the client's local operations that have already been applied to its local state, but that have yet to be committed (*i.e.*, assigned a sequence number by the server and added to the client's committed history).

SPORC synchronizes clients' local state for a particular document using the following steps, also shown in Figure 1. This section restricts its consideration to interactions with a static membership and well behaved server; we relax these restrictions in the next two sections, respectively. The flow of local operations to the server is illustrated by dashed blue arrows; the flow of operations received from the server is shown by solid red arrows.

1. A client application generates an operation, applies it to its local state immediately, and then places it at the end of the client's pending queue.
2. If the client does not currently have any operations under submission, it takes its oldest queued operation yet to be sent, *op*, assigns it a client sequence number (*clntSeqNo*), embeds in it the global sequence number of the last committed operation (*prevSeqNo*) along with the corresponding hash chain value (*prevHC*), encrypts its payload, digitally signs it, and transmits it to the server. (As an optimization, if the client has multiple operations in its pending queue, it can submit them as a single batched operation.)
3. The server adds the client-submitted *op* to its encrypted history, assigning it the next available global sequence number (*seqNo*). The server forwards *op* with this *seqNo* to all the clients participating in the document.
4. Upon receiving an encrypted operation *op*, the client verifies its signature (V) and checks that its *clntSe-*

qNo, *seqNo*, and *prevHC* fields have the expected values. If these checks succeed, the client decrypts the payload (D) for further processing. If they fail, the client concludes that the server is malicious.

5. Before adding *op* to its committed history, the client must transform it past any other operations that had been committed since *op* was generated (*i.e.*, all those with global sequence numbers greater than *op*'s *prevSeqNo*). Once *op* has been transformed, the client appends *op* to the end of the committed history.
6. If the incoming operation *op* was one that the client had initially sent, the client dequeues the oldest element in the pending queue (which will be the uncommitted version of *op*) and prepares to send its next operation. Otherwise, the client transforms *op* past all its pending operations and, conversely, transforms those operations with respect to *op*.
7. The client returns the transformed version of the incoming operation *op* to the application. The application then applies *op* to its local state.

SPORC maintains the following invariants with respect to the system's state:

Local Coherence: A client's local state is equivalent to the state it would be in if, starting with an initial empty document, it applied, in order, all of the operations in its committed history followed by all of the operations in its pending queue.

Fork* Consistency: If the server is well behaved, all clients' committed histories are linearizable (*i.e.*, for every pair of clients, one client's committed history is equal to or a prefix of the other client's committed history). If the server is faulty, however, clients' committed histories may be forked [23].

Client-Order Preservation: The order that a non-malicious server assigns to operations originating from a given client must be consistent with the order that the client assigned to those operations.

3.2 Operations

SPORC clients exchange two types of operations: *document operations*, which represent changes to the content of the document, and *meta-operations*, which represent changes to document metadata such as the document's access control list. Meta-operations are sent to the server in the clear, but the payloads of document operations are encrypted under a symmetric key that is shared among all of the clients but is unknown to the server. (See Section 4.1 for a description of how this key is chosen and distributed.) In addition, every operation is labeled with the name of the user that created it and is digitally signed by that user's private key. All operations also contain a unique client ID (*clntID*) that identifies from which of the user's client machines it came.

3.3 The Server's Limited Role

Because the SPORC server is untrusted, its role is limited to ordering and storing the operations that clients submit, most of which are encrypted. The server stores the operations in its encrypted history so that new clients joining the document or existing clients that have been disconnected can request from the server the operations that they are missing. This storage function is not essential, however, and in principle it could be handled by a different party.

Notably, since the server does not have access to the plaintext of document operations, the same generic server implementation can be used for any application that uses our protocol regardless of the kind of document being synchronized.

3.4 Sequence Numbers and Hash Chains

SPORC clients use sequence numbers and a hash chain to ensure that operations are properly serialized and that the server is well behaved. Every operation has two sequence numbers: a client sequence number (*clntSeqNo*) which is assigned by the client that submitted the operation, and a global sequence number (*seqNo*) which is assigned by the server. On receiving an operation, a client verifies that the operation's *clntSeqNo* is one greater than the last *clntSeqNo* seen from the submitting client, and that the operation's *seqNo* is one greater than the last *seqNo* that the receiving client saw. These sequence number checks enforce the "client order preservation" invariant and ensure that there are no gaps in the sequence of operations.

When a client uploads an operation op_{new} to the server, the client sets op_{new}'s *prevSeqNo* field to the global sequence number of the last committed operation, op_n, that the client knows about. The client also sets op_{new}'s *prevHC* field to the value of the client's hash chain over the committed history up to op_n. A client who receives op_{new} compares its *prevHC* with the client's own hash chain computation up to op_n. If they match, the recipient knows that its committed history is identical to the sender's committed history up to op_n, thereby guaranteeing fork* consistency.

A misbehaving server cannot modify the *prevSeqNo* or *prevHC* fields, because they are covered by the submitting client's signature on the operation. The server can try to tell two clients different global sequence numbers for the same operation, but this will cause the two clients' histories—and hence their future hash chain values—to diverge, and it will eventually be detected.

To simplify the design, each SPORC client has at most one operation "in flight" at any time: only the operation at the head of a client's pending queue can be sent to the server. Among other benefits, this rule ensures that operations' *prevSeqNo* and *prevHC* values will always refer

to operations that are in the committed history, and not to other operations that are "in flight." This restriction could be relaxed, but only at considerable cost in complexity. For similar reasons, other OT-based systems such as Google Wave adopt the same rule [44].

Prohibiting more than one in-flight operation per client is less restrictive than it might seem, as operations can be combined or batched. Like Wave, SPORC includes an application-specific composition function, which consolidates two operations into one. This can be used iteratively to combine a sequence of operations into a single one. Further, it is straightforward to batch multiple operations into a single logical operation, which is then submitted as a unit. Because operations can be composed or batched, a client can empty its pending queue every time it gets an opportunity to submit an operation to the server.

3.5 Resolving Conflicts with OT

Once a client has validated an operation received from the server, the client must use OT to resolve the conflicts that may exist between the new operation and other operations in the committed history and pending queue. These conflicts might have arisen for two reasons. First, the server may have committed additional operations since the new operation was generated. Second, the receiving client's local state might reflect uncommitted operations that reside on the client's pending queue but that other clients do not yet know about.

Before a client appends an incoming operation op_{new} to its committed history, it compares op_{new}'s *prevSeqNo* value with the global sequence number of the last committed operation. The *prevSeqNo* field indicates the last committed operation that the submitting client knew about when it uploaded op_{new}. Thus, if the values match, the client knows that no additional operations have been added to its committed history since op_{new} was generated, and the new operation can be appended directly to the committed history. But if they do not match, then other operations were committed since op_{new} was sent, and op_{new} needs to be transformed past each of them. For example, if op_{new} has a *prevSeqNo* of 10, but was assigned global sequence number 14 by the server, then the client must compute $op'_{new} \leftarrow T(op_{new}, \langle op_{11}, op_{12}, op_{13}\rangle)$ where $\langle op_{11}, op_{12}, op_{13}\rangle$ are the intervening committed operations. Only then can the resulting transformed operation op'_{new} be appended to the committed history. After appending the operation, the client updates the hash chain computed over the committed history so that future incoming operations can be validated.

At this point, if op'_{new} is one of the receiving client's own operations that it had previously uploaded to the server (or a transformed version of it), it will necessarily match the operation at the head of the pending queue. Since op'_{new} has now been committed, its uncommitted version can be retired from the pending queue, and the next pending operation can be submitted to the server. Furthermore, since the client has already optimistically applied the operation to its local state even before sending it to the server, the client does not need to apply op'_{new} again, and nothing more needs to be done.

If op'_{new} is not one of the client's own operations, however, the client must perform additional transformations in order to reestablish the "local coherence" invariant, which states that the client's local state is equal to the in-order application of its committed history followed by its pending queue. First, in order to obtain a version of op'_{new} that it can apply to its local state, the client must transform op'_{new} past all of the operations in its pending queue. This step is necessary because the pending queue contains operations that the client has already applied locally, but have not yet been committed and, therefore, were unknown to the sender of op'_{new}.

Second, the client must transform the entire pending queue with respect to op'_{new} to account for the fact that op'_{new} was appended to the committed history. More specifically, the client computes $\langle \overline{op'_1}, \ldots, \overline{op'_m}\rangle \leftarrow T(\langle \overline{op_1}, \ldots, \overline{op_m}\rangle, op'_{new})$ where $\langle \overline{op_1}, \ldots, \overline{op_m}\rangle$ is the pending queue. This transformation has the effect of pushing the pending queue forward by one operation to make room for the newly extended committed history. The operations on the pending queue need to stay ahead of the committed history because they will receive higher global sequence numbers than any of the currently committed operations. Furthermore, by transforming its unsent operations in response to updates to the document, the client reduces the amount of transformation that other clients will need to do when they eventually receive its operations.

4 Membership Management

Document membership in SPORC is controlled at the level of users, each of which is associated with a public-private key pair. When a document is first created, only the user that created it has access. Subsequently, privileged users can change the document's access control list (ACL) by submitting `ModifyUserOp` meta-operations, which get added to the document's history (covered by its hash chain), much like normal operations.

A user can be given one of three privilege levels: *reader*, which entitles the user to decrypt the document but not to submit new operations; *editor*, which entitles the user to read the document and to submit new operations (except those that change the ACL); and *administrator*, which grants the user full access, including the ability to invite new users and remove existing users. Because `ModifyUserOp`s are not encrypted, a non-malicious server will immediately reject operations from users with insufficient privileges. But because the server is untrusted,

every client maintains its own copy of the ACL, based on the history's `ModifyUserOp`s, and refuses to apply operations that came from unauthorized users.

4.1 Encrypting Document Operations

To prevent eavesdropping by the server or unapproved users, the payloads of document operations are encrypted under a symmetric key known only to the document's current members. More specifically, to create a new document, the creator generates a random AES key, encrypts it under her own public key, and then writes the encrypted key to the document's initial create meta-operation. To add new users, an administrator submits a `ModifyUserOp` that includes the document's AES key encrypted under each of the new users' public keys.

If users are removed, the AES key must be changed so that the removed users will not be able to decrypt subsequent operations. To do so, an administrator picks a new random AES key, encrypts it under the public keys of all the remaining participants, and then submits the encrypted keys as part of the `ModifyUserOp`.[3] This meta-operation also includes an encryption of the old AES key under the new AES key. This enables later users to learn earlier keys and thus decrypt old operations, without requiring the operations to be re-encrypted.

SPORC's model ensures proper access control over operations, based on how it tracks potential causality through *prevSeqNo* dependencies. Operations concurrent to a `ModifyUserOp` removal may be ordered before it and remain accessible to the user. However, once a client sees the removal meta-operation in its committed history any subsequent operation the client submits will be inaccessible to the removed user.

4.2 Barrier Operations

Concurrency also poses a challenge to membership management. Consider the situation when two clients concurrently issue `ModifyUserOp`s that both attempt to change the current symmetric key. If the server naively scheduled one after the other, then the continuous chain of old keys encrypted under new ones would be broken.

To address situations like this, we introduce a primitive called a *barrier operation*. When the server receives an operation that is marked "barrier" and assigns it global sequence number b, the server requires that every subsequent operation have a *prevSeqNo* $\geq b$. Subsequent operations that do not are rejected and must be revised and resubmitted with a later *prevSeqNo*. In this way, the server can force all future operations to depend on the barrier operation.[4]

Let us reconsider the example of two concurrent `ModifyUserOp`s, op_1 and op_2, that are marked as barriers. Suppose that the server received op_1 first and assigned it sequence number b. Since the operations were submitted concurrently, op_2's *prevSeqNo* will necessarily be less than b, and op_2 will be rejected. The client attempting to send op_2 must wait until it receives op_1, at which time it will adjust op_2 to depend on this operation before resubmitting (*i.e.*, encrypt op_1's key under its new key, and set op_2's *prevSeqNo* $\geq b$). As a result, the chain of old keys encrypted under new ones will be preserved.

Barrier operations have uses beyond membership management. For example, as described next, they are useful in implementing checkpoints on the history.

5 Extensions

This section describes extensions to the basic SPORC protocols: supporting checkpoints to reduce the size requirements for storing the committed history (Section 5.1), detecting forks through out-of-band communication (Section 5.2), and recovering from forks by replaying and possibly transforming forked operations (Section 5.3). Our current prototype does not yet implement these extensions, however.

5.1 Checkpoints

In order to reach a document's latest state, a new client in our current implementation must download and apply the entire history of committed operations. It would be more efficient for a new client to instead download a *checkpoint* of operations—a compact representation of the document's state, akin to each client's local state—and then only apply individual committed operations since the last checkpoint. Much as SPORC servers cannot transform operations, they similarly cannot perform checkpoints; SPORC once again has individual clients play this role.

To support checkpoints, each client maintains a compacted version of the committed history up to the most recent barrier operation. When a client is ready to upload a checkpoint to the server, it encrypts this compacted history under the current document key. It then creates a new `CheckpointOp` meta-operation containing the hash of the encrypted checkpoint data and submits it into the history. Requiring the checkpoint data to end in a barrier operation ensures that clients that later use the checkpoint will be able to ignore the history before the barrier without having to worry that they will need to perform OT transformations involving that old history. After all, no operation

[3] In our current implementation, the size of a `ModifyUserOp` may be linear in the number of users participating in the document, because the operation may contain the current AES key encrypted under each of the users' RSA public keys. An optimization to achieve constant-sized `ModifyUserOp`s could instead use a space-efficient broadcast encryption scheme [6].

[4] To prevent a malicious server from violating the rules governing barrier operations, an operation's "barrier" flag is covered by the operation's signature, and all clients verify that the server is handling barrier operations correctly.

after a barrier can depend on an operation before it. If the most recent barrier is too old, the client can submit a new null barrier operation before creating the checkpoint.[5]

Checkpoints raise new security challenges, however. A client that lacks the full history cannot verify the hash chain all the way back to the document's creation. It can verify that the operations it has chain together correctly, but the first operation in its history (i.e., the barrier operation) is "dangling," and its *prevHC* value cannot be verified. This is not a problem if the client knows in advance that the `CheckpointOp` is part of the valid history, but this is difficult to verify. The `CheckpointOp` will be signed by a user, and users who have access to the document are assumed to be trusted; but there must be a way to verify that the signing user had permission to access the document at the time the checkpoint was created. Unfortunately, without access to a verifiable history of individual `ModifyUserOp`s going back the beginning of the document, a client deciding whether to accept a checkpoint has no way to be certain of which users were actually members of the document at any given time.

To address these issues, we propose that the server and clients maintain a *meta-history*, alongside the committed history, that is comprised solely of meta-operations. Meta-operations are included in the committed history as before, but each one also has a *prevMetaSeqNo* pointer to a prior element of the meta-history along with a corresponding *prevMetaHC* field. Each client maintains a separate hash chain over the meta-history and performs the same consistency checks on the meta-history that it performs on the committed history.

When a client joins, before it downloads a checkpoint, it requests the entire meta-history from the server. The meta-history provides the client with a fork* consistent view of the sequence of `ModifyUserOp`s and `CheckpointOp`s that indicates whether the checkpoint's creator was an authorized user when the checkpoint was created. Moreover, the cost of downloading the entire meta-history is likely to be low because meta-operations are rare relative to document operations.

5.2 Checking for Forks Out-of-Band

Fork* consistency does not prevent a server from forking clients' state, as long as the server never tells any member of one fork about any operation done by a member of another fork. To detect such forks, clients can exchange state information out-of-band, for example, by direct socket connections, email, instant messaging, or posting on a shared server or DHT service.

Clients can exchange messages of the form $\langle c, d, s, h_s \rangle$, asserting that in client c's view of document d, the hash chain value as of sequence number s is equal to h_s. On receiving such a message, a client compares its own hash chain value at sequence number s with h_s, and if the values differ, it knows a fork has occurred. If the recipient does not yet have operations up to sequence number s, it requests them from the server; a well behaved server will always be able to supply the missing operations.

These out-of-band messages should be digitally signed to prevent forgery. To prevent out-of-band messages from leaking information about which clients are collaborating on a document, and to prevent a client from falsely claiming that it was invited into the document by a forked client, the out-of-band messages should be encrypted and MACed with a separate set of symmetric keys that are known only to nodes that have been part of the document.[6] These keys might be conveyed in the first operation of the document's history.

5.3 Recovering from a Fork

A benefit of combining OT and fork* consistency is that we can use OT to recover from forks. OT is well suited to this task because, in normal operation, OT clients are essentially creating small forks whenever they optimistically apply operations locally, and resolving these forks when they transform operations to restore consistency. In this section, we sketch an algorithm that a pair of forked clients can use to merge their divergent histories into a consistent whole. This pairwise algorithm can be repeated as necessary to resolve forks involving multiple clients, or multi-way forks.

The basic idea of the algorithm is that the two clients will abandon the malicious server and agree on a new one. Both clients will roll back their histories to their last common point before the fork, and one of them will upload the common history, up to the fork point, to the new server. Finally, each client will resubmit the operations that it saw after the fork. OT will ensure that these resubmitted operations are merged safely so that both nodes end up in the same state.

The situation becomes more complicated if the same operation appears in both histories. We cannot just remove the duplicate because later operations in the sequence may depend on it. Instead, we must cancel it out. To make this possible, we require that all operations be invertible:

[5] Having the checkpoint data end in an earlier barrier operation is better than making `CheckpointOp`s into barriers themselves. If `CheckpointOp`s were barriers, then either the client making the checkpoint would have to "lock" the history to prevent new operations from being admitted before the checkpoint was uploaded, or the system would have to reject checkpoints that did not reflect the latest state, which could potentially lead to livelock.

[6] A client falsely claiming to have been invited into the document in another fork will eventually be detected when the other clients try to recover from the (false) fork. However, this is expensive so we would prefer to avoid it. By protecting the out-of-band messages with symmetric keys known only to clients who have been in the document at some point, we reduce the set of potential liars substantially.

we must be able to construct an inverse operation op^{-1} such that applying op followed by op^{-1} results in a no-op. This is often easy to do in practice by having each operation store enough information about the prior state to determine what the inverse should be. For example, a delete operation can store the information that was deleted, enabling the creation of an insert operation as the inverse.

To cancel each duplicate, we cannot simply splice its inverse into the history right after it for the same reason that we cannot just remove the duplicate. Instead, we compute the inverse operation and then transform it past all of the operations following the duplicate. This process results in an operation that has the effect of canceling out the duplicate when appended to the end of the sequence.

6 Implementation

SPORC provides a framework for building collaborative applications that need to synchronize different kinds of state between clients. It consists of a generic server implementation and client-side libraries that implement the SPORC protocol, including the sending, receiving, encryption, and transformation of operations, as well as the necessarily consistency checks and document membership management. To build applications within the SPORC framework, a developer only needs to implement client-side functionality that (i) defines a data type for SPORC operations, (ii) defines how to transform a pair of operations, and (iii) defines how to combine multiple document operations into a single one. The server need not be modified, as it always deals with operations on encrypted data.

6.1 Variants

We implemented two variants of SPORC: a command-line version in which both client and server are stand-alone applications, and a web-based version with a browser-based client and a Java servlet. The command-line version, which we use for later microbenchmarks, is written in approximately 5500 lines of Java code (per SLOCCount [46]) and, for network communication, uses the socket-based RPC library in the open-source release of Google Wave [16]. Because the server's role is limited to ordering and storing client-supplied operations, its basic implementation is simple and only requires approximately 300 lines of code.

The web-based version shares the majority of its code with the command-line variant. The server just encapsulates the command-line server functionality in a Java servlet. The client consists almost entirely of JavaScript code that was automatically generated using the Java-to-JavaScript compiler included with the Google Web Toolkit (GWT) [12]. Network communication uses a combination of the GWT RPC framework, which wraps browser `XmlHttpRequest`s, and the GWTEventService [37], which allows the server to push messages to the browser asynchronously through a long-lived HTTP connection (the so-called "Comet" style of web programming). This prototype could be extended with HTML5's offline storage to provide disconnected operation.

The client's use cryptographic module was its only component that could not be translated to JavaScript. JavaScript remains too slow to implement public key cryptography efficiently, and browsers lack both secure storage for cryptographic keys and a secure pseudorandom number generator for key generation. To work around these limitations, we encapsulate our cryptographic module in a Java applet and implement JavaScript-to-Java communication using the LiveConnect API [28] (a strategy employed in [2, 47]). Our experience suggests it would be beneficial for browsers to provide a JavaScript API that supported basic cryptographic primitives.

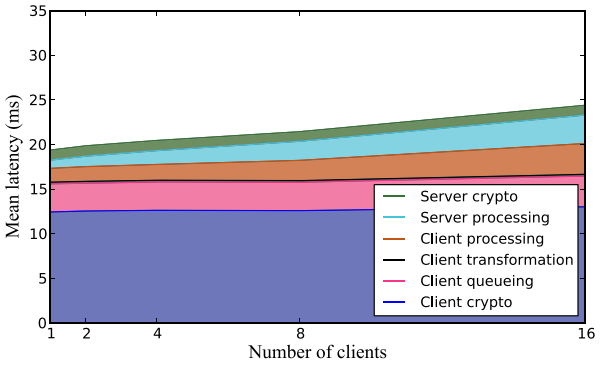
6.2 Building SPORC Applications

To demonstrate the usefulness of our framework, we built two prototype applications: a causally-consistent key-value store and a web-based collaborative text editor. The key-value store keeps a simple dictionary—mapping strings to strings—synchronized across a set of participating clients. To implement it, we defined a data type that represents a list of keys to update or remove. We wrote a simple transformation function that implements a "last writer wins" policy, as well as a composition function that merges two lists of key updates in a straightforward manner. Overall, the application-specific portion of the key-value store only required 280 lines of code.

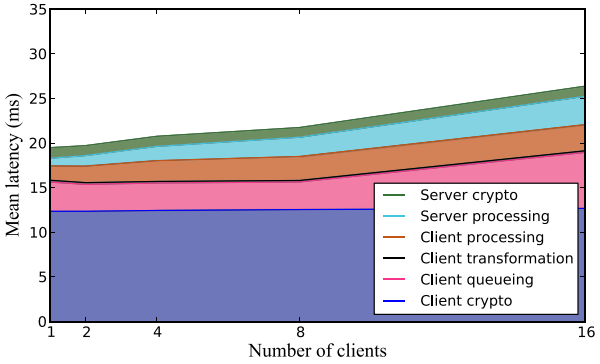
The collaborative editor allows multiple users to modify a text document simultaneously via their web browsers and see each other's changes in near real-time. It provides a user experience similar to Google Docs [14] and EtherPad [13], but, unlike those services, it does not require a trusted server. To implement it, we were able to reuse the data types and the transformation and composition functions from the open-source release of Google Wave. Although Wave is a server-centric OT system without SPORC's level of security and privacy, we were able to adapt its components for our framework with only 550 lines of wrapper code.

7 Experimental Evaluation

The user-facing collaborative applications for which SPORC was designed—*e.g.*, word processing, calendaring, and instant messaging—require latency that is low enough for human users to see each others' updates in real-time. But unlike file or storage systems, their primary goal is not high throughput. In this section, we present the results of several microbenchmarks of our Java-based

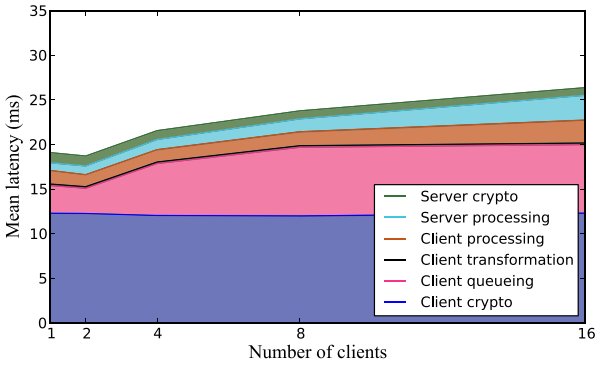


(a) Unloaded key-value store

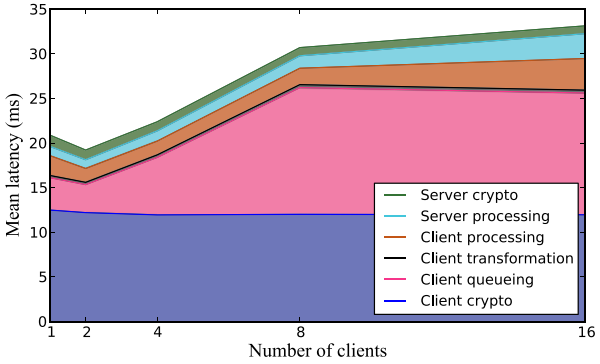


(b) Unloaded text editor

Figure 2: **Latency of SPORC with a single client writer**



(a) Loaded key-value store



(b) Loaded text editor

Figure 3: **Latency of SPORC with all clients issuing writes**

command-line version, to demonstrate SPORC's usefulness for this class of applications.

We performed our experiments on a cluster of five commodity machines, each with eight 2.3 GHz AMD Opteron cores and 8 GB of RAM, that were connected by gigabit switched Ethernet. In each of our experiments, we ran a single server instance on its own machine, along with varying numbers of client instances. To scale our system to moderate numbers of clients, in many of our experiments, we ran multiple client instances on each machine. We ran all the experiments under the OpenJDK Java VM (version IcedTea6 1.6). For RSA signatures, however, we used the Network Security Services for Java (JSS) library from the Mozilla Project [29] because, unlike Java's default cryptography library, it is implemented in native code and offers considerably better performance.

Latency. To measure SPORC's latency, we conducted three minute runs with between one and sixteen clients for both key-value and text editor operations. We tested our system under both low-load conditions, where only one of the clients submitted new operations (once every 200 ms), and high-load conditions, where all of the clients were writers. We measured latency by computing the mean time that an operation was "in flight": from the time that it was generated by the sender's application-level code, until the time it was delivered to the recipient's application.

Under low-load conditions with only one writer, we would expect the load on each client to remain constant as the number of clients increases, because each additional client does not add to the total number of operations in flight. We would, however, expect to see server latency increase modestly, as the server has to send operations to increasing numbers of clients. Indeed, as shown in Figure 2, the latency due to server processing increased from under 1 ms with one client to over 3 ms with sixteen clients, while overall latency increased modestly from approximately 19 ms to approximately 25 ms.[7]

On the other hand, when every client is a writer, we would expect the load on each client to increase with the number of clients. As expected, Figure 3 shows that with sixteen clients under loaded conditions, overall latency is higher: approximately 26 ms for key-value operations and 33 ms for the more expensive text-editor operations. The biggest contributor to this increase is client queueing, which is primarily the time that a client's received operations spend in its incoming queue before being processed. Queueing delay begins at around 3 ms for one

[7]Figure 2 also shows small increases in the latency of client processing and queuing when the number of clients was greater than four. These increases are most likely due to the fact that, when we conducted experiments with more than four clients, we ran multiple client instances per machine.

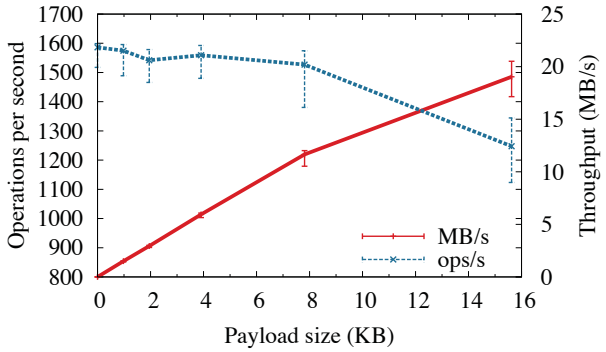


Figure 4: **Server throughput as a function of payload size.**

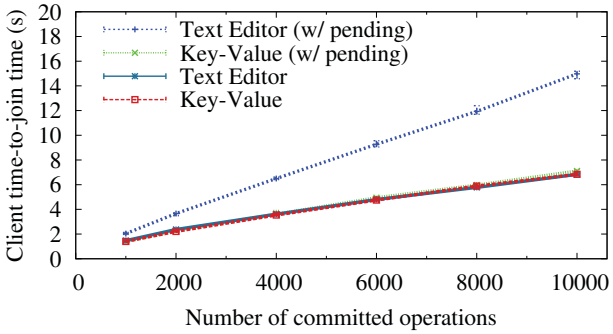


Figure 5: **Client time-to-join given a variable length history**

client and then increases steadily until it levels off at approximately 8 ms for the key-value application and 14 ms for the text editor. Despite this increase, Figure 3 demonstrates that SPORC successfully supports real-time collaboration for moderately-sized groups, even under load. As these experiments were performed on a local-area network, a wide-area deployment of SPORC would see an increase in latency that reflects the correspondingly higher network round-trip-time.

Figures 2 and 3 also show that client-side cryptographic operations account for a large share of overall latency. This occurs because SPORC performs a 2048-bit RSA signature on every outgoing operation and because Mozilla JSS, while better than Java's cryptography built-in library, still requires about 10 ms to compute a single signature. Using an optimized implementation of a more efficient signature scheme, such as ESIGN, could improve the latency of signatures by nearly two orders of magnitude [24].

Server throughput. We measured the server's maximum throughput by saturating the server with operations using 100 clients. These particular clients were modified to allow them to have more than one operation in flight at a time. Figure 4 shows server throughput as a function of payload size, measured in terms of both operations per second and MB per second. Each data point was computed by performing a three minute run of the system and then taking the median of the mean throughput of each one second interval. The error bars represent the 5th and 95th percentiles. The figure shows that, as expected, when payload size increases, the number of operations per second decreases, because each operation requires more time to process. But, at the same time, data throughput (MB/s) increases, because the processing overhead per byte decreases.

Client time-to-join. Because our current implementation lacks the checkpoints of Section 5.1, when a client joins the document, it must first download each individual operation in the committed history. To evaluate the cost of joining an existing document, we first filled the history with varying numbers of operations. Then, we measured the time it took for a new client to receive the shared decryption key and download and process all of the committed operations. We performed two kinds of experiments: one where the client started with an empty local state, and a second in which the client had 2000 pending operations that had yet to be submitted to the server. The purpose of the second test was to measure how long it would take for a client that had been working offline for some length of time to synchronize with the current state of the document. Synchronization requires the client to transform its pending operations past the committed operations that the client has not seen; thus, it is more costly than joining a document with an empty local state. Notably, since the-fork recovery algorithm sketched in Section 5.3 relies on the same mechanism that is used to synchronize clients that have been offline—it treats operations after the fork as if they were pending uncommitted operations—this test also sheds light on the cost of recovering from a fork.

Figure 5 shows time-to-join as a function of history size. Each data point represents the median of ten runs, and the error bars correspond to the 10th and 90th percentiles. We find that time-to-join is linear in the number of committed operations. It takes a client with an empty local state approximately one additional second to join a document for every additional 1000 committed operations.

In addition, the figure shows that the time-to-join with a significant number of pending operations varies greatly by application. In the key-value application, the transformation function is cheap, because it is effectively a no-op if the given operations do not affect the same keys. As a result, the cost of transforming 2000 operations adds little to the time-to-join. By contrast, the text editor's more complex transformation function adds a non-trivial, although still acceptable, amount of overhead.

8 Related Work

Real-time "groupware" collaboration systems have adapted classic distributed systems techniques for time-stamping and ordering (*e.g.*, [4, 5, 20]), but have also introduced novel techniques to automatically resolve

conflicts between concurrent operations in an intention-preserving manner (*e.g.*, [11, 18, 33, 38, 39, 40, 41, 42]). These techniques form the basis of SPORC's client synchronization mechanism and allow it to support slow or disconnected networks. Several systems also use OT to implement undo functionality (*e.g.*, [32, 33]), and SPORC's fork recovery algorithm draws upon these approaches. Furthermore, as an alternative to OT, Bayou [43] allows applications to specify conflict detection and merge protocols to reconcile concurrent operations. Most of these protocols focus on decentralized settings and use n-way reconciliation, but several well-known systems use a central server to simplify synchronization between clients (including Jupiter [30] and Google Wave [44]). SPORC also uses a central server for ordering and storage, but allows the server to be untrusted. Secure Spread [3] presents several efficient message encryption and key distribution architectures for such client-server group collaboration settings. But unlike SPORC, it relies on trusted servers that can generate keys and re-encrypt messages as needed.

Traditionally, distributed systems have defended against potentially malicious servers by replicating functionality and storage over multiple servers. Protocols, such as Byzantine fault tolerant (BFT) replicated state machines [9, 21, 48] or quorum systems [1, 26], can then guarantee safety and liveness, provided that some fraction of these servers remain non-faulty. Modern approaches optimize performance by, for example, concurrently executing independent operations [19], permitting client-side speculation [45], or supporting eventual consistency [35]. BFT protocols face criticism, however, because when the number of correct servers falls below a certain threshold (typically two-thirds), they cannot make progress.

Subsequently, variants of fork consistency protocols (*e.g.*, [7, 27, 31]) have addressed the question of how much safety one can achieve with a single untrusted server. These works demonstrate that server *equivocation* can always be detected unless the server permanently forks the clients into groups that cannot communicate with each other. SUNDR [24] and FAUST [8] use these fork consistency techniques to implement storage protocols on top of untrusted servers. Other systems, such as A2M [10] and TrInc [22], rely on trusted hardware to detect server equivocation. BFT2F [23] combines techniques from BFT replication and SUNDR to achieve fork* consistency with higher fractions of faulty nodes than BFT can resist. SPORC borrows from the design of BFT2F in its use of hash chains to limit equivocation, but unlike BFT2F or any of these other systems, SPORC allows disconnected operation and enables clients to recover from server equivocation, not just detect it.

Like SPORC, two very recent systems, Venus [34] and Depot [25], allow clients to use a cloud resource without having to trust it, and they also support some degree of disconnected operation. Venus provides strong consistency in the face of a potentially malicious server, but does not support applications other than key-value storage. Furthermore, unlike SPORC, it requires the majority of a "core set" of clients to be online in order to achieve most of its consistency guarantees. In addition, although members may be added dynamically to the group editing the shared state, it does not allow access to be revoked, nor does it provide a mechanism for distributing encryption keys. Depot, on the other hand, does not rely on the availability of a "core set" of clients and supports varied applications. Moreover, similar to SPORC, it allows clients to recover from malicious forks using the same mechanism that it uses to keep clients synchronized. But rather than providing a means for reconciling conflicting operations as SPORC does with OT, Depot relies on the application for conflict resolution. Because Depot treats clients and servers identically, it can also tolerate faulty clients, in addition to faulty servers. Unlike SPORC, however, Depot does not consider dynamic access control or confidentiality.

9 Conclusion

Our original goal for SPORC was to design a general framework for web-based group collaboration that could leverage cloud resources, but not be beholden to them for privacy guarantees. This goal leads to a design in which servers only store encrypted data, and each client maintains its own local copy of the shared state. But when each client has its own copy of the state, the system must keep them synchronized, and operational transformation provides a way do to so. OT enables optimistic updates and automatically reconciles clients' conflicting states.

Supporting applications that need timely commits requires a central server. But if we do not trust the server to preserve data privacy, we should not trust it to commit operations correctly either. This requirement led us to employ fork* consistency techniques to allow clients to detect server equivocation about the order of committed operations. But beyond the benefits that each provides independently, this work shows that OT and fork* consistency complement each other well. Whereas prior systems that enforced fork* consistency alone were only able to detect malicious forks, by combining fork* consistency with OT, SPORC can recover from them using the same mechanism that keeps clients synchronized.

In addition to these conceptual contributions, we present a membership management architecture that provides dynamic access control and key distribution with an untrusted server, even in the face of concurrency. Finally, we also demonstrate the flexibility of our design by implementing two applications: a causally-consistent key-value store and a browser-based collaborative text editor.

Acknowledgments. We thank Siddhartha Sen, Jinyuan Li, Alma Whitten, Alexander Shraer, and Christian Cachin for their insights. We also thank our shepherd, Lidong Zhou, and the anonymous reviewers for their helpful comments. This research was supported by funding from Google and the NSF CAREER grant CNS-0953197.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proc. SOSP*, Oct. 2005.

[2] B. Adida. Helios: Web-based open-audit voting. In *Proc. USENIX Security*, Aug. 2008.

[3] Y. Amir, C. Nita-rotaru, J. Stanton, and G. Tsudik. Secure spread: An integrated architecture for secure group communication. *IEEE Trans. Dependable and Secure Computing*, 2:248–261, 2005.

[4] P. Bernstein, N. Goodman, and V. Hadzilacos. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comp. Systems*, 9(3): 272–314, Aug. 1991.

[6] D. Boneh, C. Gentry, and B. Waters. Collusion resistant broadcast encryption with short ciphertexts and private keys. In *Advances in Cryptology – CRYPTO*, Aug. 2005.

[7] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proc. PODC*, Aug. 2007.

[8] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *Proc. Dependable Systems and Networks (DSN)*, June 2009.

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. OSDI*, Feb. 1999.

[10] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. SOSP*, Oct. 2007.

[11] C. Ellis and S. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Record*, 18(2):399–407, 1989.

[12] Google. Google Web Toolkit (GWT). `http://code.google.com/webtoolkit/`, 2010.

[13] Google. EtherPad. `http://etherpad.com/`, 2010.

[14] Google. Google Docs. `http://docs.google.com/`, 2010.

[15] Google. Government requests directed to Google and YouTube. `http://www.google.com/governmentrequests/`, 2010.

[16] Google. Google Wave federation protocol. `http://code.google.com/p/wave-protocol/`, 2010.

[17] M. Handley and J. Crowcroft. Network text editor (NTE): A scalable shared text editor for MBone. In *Proc. SIGCOMM*, Oct. 1997.

[18] A. Karsenty and M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proc. ICDCS*, May 1993.

[19] R. Kotla and M. Dahlin. High-throughput Byzantine fault tolerance. In *Proc. Dependable Systems and Networks (DSN)*, June 2004.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, 1978.

[21] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Programming Language Systems*, 4(3), 1982.

[22] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proc. NSDI*, Apr. 2009.

[23] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proc. NSDI*, Apr. 2007.

[24] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. OSDI*, Dec. 2004.

[25] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proc. OSDI*, Oct. 2010.

[26] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. STOC*, May 1997.

[27] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Proc. PODC*, July 2002.

[28] Mozilla Project. LiveConnect. `https://developer.mozilla.org/en/LiveConnect`, 2010.

[29] Mozilla Project. Network security services for Java (JSS). `https://developer.mozilla.org/En/JSS`, 2010.

[30] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *Proc. UIST*, Nov. 1995.

[31] A. Oprea and M. K. Reiter. On consistency of encrypted files. In *Proc. Symposium on Distributed Computing (DISC)*, Sept. 2006.

[32] A. Prakash and M. Knister. A framework for undoing actions in collaborative systems. *ACM Trans. Computer-Human Interaction*, 4(1):295–330, Dec. 1994.

[33] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proc. CSCW*, Nov. 1996.

[34] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proc. ACM CCSW*, Oct. 2010.

[35] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: eventually consistent byzantine-fault tolerance. In *Proc. NSDI*, Apr. 2009.

[36] M. Stonebraker. The case for shared nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, 1986.

[37] S. Strohschein. GWTEventService. `http://code.google.com/p/gwteventservice/`, 2010.

[38] M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in distributed collaborative environment. In *Proc. Conf. Supporting Group Work (GROUP)*, Nov. 1997.

[39] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. CSCW*, Nov. 1998.

[40] C. Sun, X. Jia, Y. Yang, and Y. Zhang. A generic operation transformation schema for consistency maintenance in realtime cooperative editing systems. In *Proc. Conf. Supporting Group Work (GROUP)*, Nov. 1997.

[41] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Computer-Human Interaction*, 5(1):64–108, 1998.

[42] D. Sun, S. Xia, C. Sun, and D. Chen. Operational transformation for collaborative word processing. In *Proc. CSCW*, Nov. 2004.

[43] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. SOSP*, Dec. 1995.

[44] D. Wang and A. Mah. Google wave operational transformation. `http://www.waveprotocol.org/whitepapers/operational-transform`, Apr. 2010.

[45] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *Proc. NSDI*, Apr. 2009.

[46] D. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`, 2010.

[47] T. D. Wu. The secure remote password protocol. In *Proc. NDSS*, Mar. 1998.

[48] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. SOSP*, Oct. 2003.

Onix: A Distributed Control Platform for Large-scale Production Networks

Teemu Koponen*, Martin Casado*, Natasha Gude*, Jeremy Stribling*, Leon Poutievski†,
Min Zhu†, Rajiv Ramanathan†, Yuichiro Iwata‡, Hiroaki Inoue‡, Takayuki Hama‡, Scott Shenker§

Abstract

Computer networks lack a general control paradigm, as traditional networks do not provide any network-wide management abstractions. As a result, each new function (such as routing) must provide its own state distribution, element discovery, and failure recovery mechanisms. We believe this lack of a common control platform has significantly hindered the development of flexible, reliable and feature-rich network control planes.

To address this, we present Onix, a platform on top of which a network control plane can be implemented as a distributed system. Control planes written within Onix operate on a global view of the network, and use basic state distribution primitives provided by the platform. Thus Onix provides a general API for control plane implementations, while allowing them to make their own trade-offs among consistency, durability, and scalability.

1 Introduction

Network technology has improved dramatically over the years with line speeds, port densities, and performance/price ratios all increasing rapidly. However, network control plane mechanisms have advanced at a much slower pace; for example, it takes several years to fully design, and even longer to widely deploy, a new network control protocol.[1] In recent years, as new control requirements have arisen (*e.g.*, greater scale, increased security, migration of VMs), the inadequacies of our current network control mechanisms have become especially problematic. In response, there is a growing movement, driven by both industry and academia, towards a control paradigm in which the control plane is decoupled from the forwarding plane and built as a distributed system.[2]

In this model, a network-wide control platform, running on one or more servers in the network, oversees a set of simple switches. The control platform handles state distribution – collecting information from the switches and distributing the appropriate control state to them, as well as coordinating the state among the various platform servers – and provides a programmatic interface upon which developers can build a wide variety of management applications. (The term "management application" refers to the control logic needed to implement management features such as routing and access control.)[3] For the purposes of this paper, we refer to this paradigm for network control as *Software-Defined Networking* (SDN).

This is in contrast to the traditional network control model in which state distribution is limited to link and reachability information and the distribution model is fixed. Today a new network control function (*e.g.*, scalable routing of flat intra-domain addresses [21]) requires its own distributed protocol, which involves first solving a hard, low-level design problem and then later overcoming the difficulty of deploying this design on switches. As a result, networking gear today supports a baroque collection of control protocols with differing scalability and convergence properties. On the other hand, with SDN, a new control function requires writing control logic on top of the control platform's higher-level API; the difficulties of implementing the distribution mechanisms and deploying them on switches are taken care of by the platform. Thus, not only is the work to implement a new control function reduced, but the platform provides a unified framework for understanding the scaling and performance properties of the system.

Said another way, the essence of the SDN philosophy is that basic primitives for state distribution should be implemented once in the control platform rather than separately for individual control tasks, and should use well-known and general-purpose techniques from the distributed systems literature rather than the more specialized algorithms found in routing protocols and other network control mechanisms. The SDN paradigm allows network system implementors to use a single control platform to implement a range of control functions (*e.g.*, routing, traffic engineering, access control, VM migration) over a spectrum of control granularities (from individual flows to large traffic aggregates) in a variety of contexts (*e.g.*, enterprises, datacenters, WANs).

*Nicira Networks
†Google
‡NEC
§International Computer Science Institute (ICSI) & UC Berkeley

[1] See, for example, TRILL [32], a recent success story which has been in the design and specification phase for over 6 years.

[2] The industrial efforts in this area are typically being undertaken by entities that operate large networks, not by the incumbent networking equipment vendors themselves.

[3] Just to be clear, we only imagine a single "application" being used in any particular deployment; this application might address several issues, such as routing and access control, but the control platform is not designed to allow multiple applications to control the network simultaneously (unless the network is "physically sliced" [28]).

Because the control platform simplifies the duties of both switches (which are controlled by the platform) and the control logic (which is implemented on top of the platform) while allowing great generality of function, the control platform is the crucial enabler of the SDN paradigm. The most important challenges in building a production-quality control platform are:

- *Generality:* The control platform's API must allow management applications to deliver a wide range of functionality in a variety of contexts.
- *Scalability:* Because networks (particularly in the datacenter) are growing rapidly, any scaling limitations should be due to the inherent problems of state management, not the implementation of the control platform.
- *Reliability:* The control platform must handle equipment (and other) failures gracefully.
- *Simplicity:* The control platform should simplify the task of building management applications.
- *Control plane performance:* The control platform should not introduce significant additional control plane latencies or otherwise impede management applications (note that forwarding path latencies are unaffected by SDN). However, the requirement here is for adequate control-plane performance, not optimal performance. When faced with a tradeoff between generality and control plane performance, we try to optimize the former while satisficing the latter.[4]

While a number of systems following the basic paradigm of SDN have been proposed, to date there has been little published work on how to build a network control platform satisfying all of these requirements. To fill this void, in this paper we describe the design and implementation of such a control platform called Onix (Sections 2-5). While we do not yet have extensive deployment experience with Onix, we have implemented several management applications which are undergoing production beta trials for commercial deployment. We discuss these and other use cases in Section 6, and present some performance measures of the platform itself in Section 7.

Onix did not arise *de novo*, but instead derives from a long history of related work, most notably the line of research that started with the 4D project [15] and continued with RCP [3], SANE [6], Ethane [5] and NOX [16] (see [4,23] for other related work). While all of these were steps towards shielding protocol design from low-level details, only NOX could be considered a control platform offering a general-purpose API.[5] However, NOX did not adequately address reliability, nor did it give the application designer enough flexibility to achieve scalability.

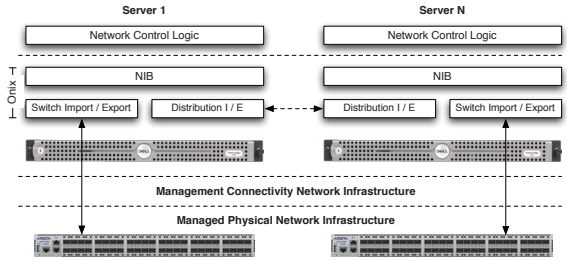


Figure 1: There are four components in an Onix controlled network: managed physical infrastructure, connectivity infrastructure, Onix, and the control logic implemented by the management application. This figure depicts two Onix instances coordinating and sharing (via the dashed arrow) their views of the underlying network state, and offering the control logic a read/write interface to that state. Section 2.2 describes the NIB.

The primary contributions of Onix over existing work are thus twofold. First, Onix exposes a far more general API than previous systems. As we describe in Section 6, projects being built on Onix are targeting environments as diverse as the WAN, the public cloud, and the enterprise data center. Second, Onix provides flexible distribution primitives (such as DHT storage and group membership) allowing application designers to implement control applications without re-inventing distribution mechanisms, and while retaining the flexibility to make performance/scalability trade-offs as dictated by the application requirements.

2 Design

Understanding how Onix realizes a production-quality control platform requires discussing two aspects of its design: the context in which it fits into the network, and the API it provides to application designers.

2.1 Components

There are four components in a network controlled by Onix, and they have very distinct roles (see Figure 1).

- *Physical infrastructure:* This includes network switches and routers, as well as any other network elements (such as load balancers) that support an interface allowing Onix to read and write the

[4]There might be settings where optimizing control plane performance is crucial. For example, if one cannot use backup paths for improved reliability, one can only rely on a fine-tuned routing protocol. In such settings one might not use a general-purpose control platform, but instead adopt a more specialized approach. We consider such settings increasingly uncommon.

[5]Only a brief sketch of NOX has been published; in some ways, this paper can be considered the first in-depth discussion of a NOX-like design, albeit in a second-generation form.

state controlling the element's behavior (such as forwarding table entries). These network elements need not run any software other than that required to support this interface and (as described in the following bullet) achieve basic connectivity.

- *Connectivity infrastructure:* The communication between the physical networking gear and Onix (the "control traffic") transits the connectivity infrastructure. This control channel may be implemented either in-band (in which the control traffic shares the same forwarding elements as the data traffic on the network), or out-of-band (in which a separate physical network is used to handle the control traffic). The connectivity infrastructure must support bidirectional communication between the Onix instances and the switches, and optionally supports convergence on link failure. Standard routing protocols (such as IS-IS or OSPF) are suitable for building and maintaining forwarding state in the connectivity infrastructure.

- *Onix:* Onix is a distributed system which runs on a cluster of one or more physical servers, each of which may run multiple Onix instances. As the control platform, Onix is responsible for giving the control logic programmatic access to the network (both reading and writing network state). In order to scale to very large networks (millions of ports) and to provide the requisite resilience for production deployments, an Onix instance is also responsible for disseminating network state to other instances within the cluster.

- *Control logic:* The network control logic is implemented on top of Onix's API. This control logic determines the desired network behavior; Onix merely provides the primitives needed to access the appropriate network state.

These are the four basic components of an SDN-based network. Before delving into the design of Onix, we should clarify our intended range of applicability. We assume that the physical infrastructure can forward packets much faster (typically by two or more orders of magnitude) than Onix (or any general control platform) can process them; thus, we do not envision using Onix to implement management functions that require the control logic to know about per-packet (or other rapid) changes in network state.

2.2 The Onix API

The principal contribution of Onix is defining a *useful* and *general* API for network control that allows for the development of scalable applications. Building on previous work [16], we designed Onix's API around a view of the physical network, allowing control applications to read and write state to any element in the network. Our API is therefore data-centric, providing methods for keeping state consistent between the in-network elements and the control application (running on multiple Onix instances).

More specifically, Onix's API consists of a data model that represents the network infrastructure, with each network element corresponding to one or more data objects. The control logic can: read the current state associated with that object; alter the network state by operating on these objects; and register for notifications of state changes to these objects. In addition, since Onix must support a wide range of control scenarios, the platform allows the control logic to customize (in a way we describe later) the data model and have control over the placement and consistency of each component of the network state.

The copy of the network state tracked by Onix is stored in a data structure we call the Network Information Base (NIB), which we view as roughly analogous to the Routing Information Base (RIB) used by IP routers. However, rather than just storing prefixes to destinations, the NIB is a graph of all network entities within a network topology. The NIB is both the heart of the Onix control model and the basis for Onix's distribution model. Network control applications are implemented by reading and writing to the NIB (for example modifying forwarding state or accessing port counters), and Onix provides scalability and resilience by replicating and distributing the NIB between multiple running instances (as configured by the application).

While Onix handles the replication and distribution of NIB data, it relies on application-specific logic to both detect and provide conflict resolution of network state as it is exchanged between Onix instances, as well as between an Onix instance and a network element. The control logic may also dictate the consistency guarantees for state disseminated between Onix instances using distributed locking and consensus algorithms.

In order to simplify the discussion, we assume that the NIB only contains physical entities in the network. However, in practice it can easily be extended to support logical elements (such as tunnels).

2.3 Network Information Base Details

At its most generic level, the NIB holds a collection of *network entities*, each of which holds a set of key-value pairs and is identified by a flat, 128-bit, global identifier. These network entities are the base structure from which all types are derived. Onix supports stronger typing through *typed entities*, representing different network elements (or their subparts). Typed entities then contain a predefined set of attributes (using the key-value pairs) and methods to perform operations over those attributes.

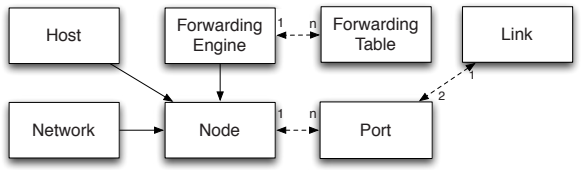


Figure 2: The default network entity classes provided by Onix's API. Solid lines represent inheritance, while dashed lines correspond to referential relation between entity instances. The numbers on the dashed lines show the quantitative mapping relationship (*e.g.*, one `Link` maps to two `Ports`, and two `Ports` can map to the same `Link`). Nodes, ports and links constitute the network topology. All entity classes inherit the same base class providing generic key-value pair access.

For example, there is a `Port` entity class that can belong to a list of ports in a `Node` entity. Figure 2 illustrates the default set of typed entities Onix provides – all typed entities have a common base class limited to generic key-value pair access. The type-set within Onix is not fixed and applications can subclass these basic classes to extend Onix's data model as needed.[6]

The NIB provides multiple methods for the control logic to gain access to network entities. It maintains an index of all of its entities based on the entity identifier, allowing for direct querying of a specific entity. It also supports registration for notifications on state changes or the addition/deletion of an entity. Applications can further extend the querying capabilities by listening for notifications of entity arrivals and maintaining their own indices.

The control logic for a typical application is therefore fairly straightforward. It will register to be notified on some state change (*e.g.*, the addition of new switches and ports), and once the notification fires, it will manipulate the network state by modifying the key-value pairs of the affected entities.

The NIB provides neither fine-grained nor distributed locking mechanisms, but rather a mechanism to request and release exclusive access to the NIB data structure of the local instance. While the application is given the guarantee that no other thread is updating the NIB within the same controller instance, it is not guaranteed the state (or related state) remains untouched by other Onix instances or network elements. For such coordination, it must use mechanisms implemented externally to the NIB. We describe this in more detail in Section 4; for now, we assume this coordination is mostly static and requires control logic involvement during failure conditions.

All NIB operations are asynchronous, meaning that updating a network entity only guarantees that the update message will eventually be sent to the corresponding network element and/or other Onix instances – no ordering or latency guarantees are given. While this has the potential to simplify the control logic and make multiple modifications more efficient, often it is useful to know when an update has successfully completed. For instance, to minimize disruption to network traffic, the application may require the updating of forwarding state on multiple switches to happen in a particular order (to minimize, for example, packet drops). For this purpose, the API provides a synchronization primitive: if called for an entity, the control logic will receive a callback once the state has been pushed. After receiving the callback, the control logic may then inspect the contents of the NIB and verify that the state is as expected before proceeding. We note that if the control logic implements distributed coordination, race-conditions in state updates will either not exist or will be transient in nature.

An application may also only rely on NIB notifications to react to failures in modifications as they would any other network state changes. Table 1 lists available NIB-manipulation methods.

Category	Purpose
Query	Find entities.
Create, destroy	Create and remove entities.
Access attributes	Inspect and modify entities.
Notifications	Receive updates about changes.
Synchronize	Wait for updates being exported to network elements and controllers.
Configuration	Configure how state is imported to and exported from the NIB.
Pull	Ask for entities to be imported on-demand.

Table 1: Functions provided by the Onix NIB API.

3 Scaling and Reliability

To be a viable alternative to the traditional network architecture, Onix must meet the scalability and reliability requirements of today's (and tomorrow's) production networks. Because the NIB is the focal point for the system state and events, its use largely dictates the scalability and reliability properties of the system. For example, as the number of elements in the network increases, a NIB that is not distributed could exhaust system memory. Or, the number of network events (generated by the NIB) or work required to manage them could grow to saturate the CPU of a single Onix instance.[7]

This and the following section describe the NIB distribution framework that enables Onix to scale to very

[6] Subclassing also enables control over how the key-value pairs are stored within the entity. Control logics may prefer different trade-offs between memory and CPU usage.

[7] In one of our upcoming deployments, if a single-instance application took one second to analyze the statistics of a single `Port` and compute a result (*e.g.*, for billing purposes), that application would take two months to process all `Ports` in the NIB.

large networks, and to handle network and controller failure.

3.1 Scalability

Onix supports three strategies that can used to improve scaling. First, it allows control applications to *partition* the workload so that adding instances reduces work without merely replicating it. Second, Onix allows for *aggregation* in which the network managed by a cluster of Onix nodes appears as a single node in a separate cluster's NIB. This allows for federated and hierarchical structuring of Onix clusters, thus reducing the total amount of information required within a single Onix cluster. Finally, Onix provides applications with control over the *consistency* and *durability* of the network state. In more detail:

- *Partitioning.* The network control logic may configure Onix so that a particular controller instance keeps only a subset of the NIB in memory and up-to-date. Further, one Onix instance may have connections to a subset of the network elements, and subsequently, can have fewer events originating from the elements to process.

- *Aggregation.* In a multi-Onix setup, one instance of Onix can expose a subset of the elements in its NIB as an aggregate element to another Onix instance. This is typically used to expose less fidelity to upper tiers in a hierarchy of Onix controllers. For example, in a large campus network, each building might be managed by an Onix controller (or controller cluster) which exposes all of the network elements in that building as a single aggregate node to a global Onix instance which performs campus-wide traffic engineering. This is similar in spirit to global control management paradigms in ATM networks [27].

- *Consistency and durability.* The control logic dictates the consistency requirements for the network state it manages. This is done by implementing any of the required distributed locking and consistency algorithms for state requiring strong consistency, and providing conflict detection and resolution for state not guaranteed to be consistent by use of these algorithms. By default, Onix provides two data stores that an application can use for state with differing preferences for durability and consistency. For state applications that favor durability and stronger consistency, Onix offers a replicated transactional database and, for volatile state that is more tolerant of inconsistencies, a memory-based one-hop DHT. We return to these data stores in Section 4.

The above scalability mechanisms can be used to manage networks too large to be controlled by a single Onix instance. To demonstrate this, we will use a running example: an application that can establish paths between switches in a managed topology, with the goal of establishing complete routes through the network.

Partition. We assume a network with a modest number of switches that can be easily handled by a single Onix instance. However, the number and size of all forwarding state entries on the network exceeds the memory resources of a single physical server.

To handle such a scenario, the control logic can replicate all switch state, but it must partition the forwarding state and assign each partition to a unique Onix instance responsible for managing that state. The method of partitioning is unimportant as long as it creates relatively consistent chunks.

The control logic can record the switch and link inventory in the fully-replicated, durable state shared by all Onix instances, and it can coordinate updates using mechanisms provided by the platform. However, information that is more volatile, such as link utilization levels, can be stored in the DHT. Each controller can use the NIB's representation of the complete physical topology (from the replicated database), coupled with link utilization data (from the DHT), to configure the forwarding state necessary to ensure paths meeting the deployment's requirements throughout the network.

The resulting distribution strategy closely resembles the use of head-end routers in MPLS [24] to manage tunnels. However, instead of a DHT, MPLS uses intra-domain routing protocols to disseminate the link utilization information.

Aggregate. As our example network grows, partitioning the path management no longer suffices. We assume that the Onix instances are still capable of holding the full NIB, but the control logic cannot keep up with the number of network events and thus saturates the CPU. This scenario follows from our experience in which CPU is commonly the limiting factor for control applications.

To shield remote instances from high-rates of updates, the application can aggregate a topology as a single logical node, and use that as the unit of event dissemination between instances. For example, the topology can be divided into logical areas, each managed by a distinct Onix instance. Onix instances external to an area would know the exact physical topology within the area, but would retrieve only topologically-aggregated link-utilization information from the DHT (originally generated by instances within that area).

This use of topological aggregation is similar to ATM PNNI [27], in which the internals of network areas are aggregated into single logical nodes when exposed to neighboring routers. The difference is that the Onix instances and switches still have full connectivity between

them and it is assumed that the latency between any element (between the switches and Onix instances or between Onix instances) is not a problem.

Partition further. At some point, the number of elements within a control domain will overwhelm the capacity of a single Onix instance. However, due to relatively slow change rates of the physical network, it is still possible to maintain a distributed view of the network graph (the NIB).

Applications can still rely on aggregating link utilization information, but in a partitioned NIB scheme, they would use the inter-Onix state distribution mechanisms to mediate requests to switches in remote areas; this can be done by using NIB attributes as a remote communication channel. The "request" and "response" are relayed between the areas using the DHT. Because this transfer might happen via a third Onix instance, any application that needs faster response times may configure DHT key ranges for areas and use DHT keys such that for the modified entity its attributes are stored within the proper area.

It is possible for this approach to scale to wide-area deployment scenarious. For example, each partition could represent a large network area, and each network is exposed as an aggregate node to a cluster of Onix instances that make global routing decisions over the aggregate nodes. Thus, each partition makes local routing decisions, and the cluster makes routing decisions between these partitions (abstracting each as a single logical node). The state distribution requirements for this approach would be almost identical to hierarchical MPLS.

Inter-domain aggregation. Once the controlled network spans two separate ASes, sharing full topology information among the Onix instances becomes infeasible due to privacy reasons and the control logic designer needs to adapt the design again to changed requirements.

The platform does not dictate how the ASes would peer, but at a high-level they would have two requirements to fulfill: *a)* sharing their topologies at some level of detail (while preserving privacy) with their peers, and *b)* establishing paths for each other proactively (according to a peering contract) or on-demand, and exchanging their ingress information. For both requirements, there are proposals in academia [13] and industry deployments [12] that applications could implement to arrange peering between Onix instances in adjacent ASes.

3.2 Reliability

Control applications on Onix must handle four types of network failures: forwarding element failures, link failures, Onix instance failures, and failures in connectivity between network elements and Onix instances (and between the Onix instances themselves). This section discusses each in turn.

Network element and link failures. Modern control planes already handle network element and link failures, and control logic built on Onix can use the same mechanisms. If a network element or link fails, the control logic has to steer traffic around the failures. The dissemination times of the failures through the network together with the re-computation of the forwarding tables define the minimum time for reacting to the failures. Given increasingly stringent requirements convergence times, it may be preferrable that convergence be handled partially by backup paths with fast failover mechanisms in the network element.

Onix failures. To handle an Onix instance failure, the control logic has two options: running instances can detect a failed node and take over the responsibilities of the failed instance quickly, or more than one instance can simultaneously manage each network element.

Onix provides coordination facilities (discussed in Section 4) for detecting and reacting to instance failures. For the simultaneous management of a network element by more than one Onix instance, the control logic has to handle lost update race conditions when writing to network state. To help, Onix provides hooks that applications can use to determine whether conflicting changes made by other instances to the network element can be overridden. Provided the control logic computes the same network element state in a deterministic fashion at each Onix instance, *i.e.*, every Onix instance implements the same algorithm, the state can remain inconsistent only transiently. At the high-level, this approach is similar to the reliability mechanisms of RCP [3], in which multiple centralized controllers push updates over iBGP to edge routers.

Connectivity infrastructure failures. Onix state distribution mechanisms decouple themselves from the underlying topology. Therefore, they require connectivity to recover from failures, both between network elements and Onix instances as well as between Onix instances. There are a number of methods for establishing this connectivity. We describe some of the more common deployment scenarios below.

It is not unusual for an operational network to have a dedicated physical network or VLAN for management. This is common, for example, in large datacenter buildouts or hosting environments. In such environments, Onix can use the management network for control traffic, isolating it from forwarding plane disruptions. Under this deployment model, the control network uses standard networking gear and thus any disruption to the control network is handled with traditional protocols (*e.g.*, OSPF or spanning tree).

Even if the environment does not provide a separate control network, the physical network topology is typically known to Onix. Therefore, it is possible for the control logic to populate network elements with static forwarding state to establish connecitivty between Onix and the switches. To guarantee connectivity in presence of failures, source routing can be combined with multi-pathing (also implemented below Onix): source routing packets over multiple paths can guarantee extremely reliable connectivity to the managed network elements, as well as between Onix instances.

4 Distributing the NIB

This section describes how Onix distributes its Network Information Base and the consistency semantics an application can expect from it.

4.1 Overview

Onix's support for state distribution mechanisms was guided by two observations on network management applications. First, applications have differing requirements on scalability, frequency of updates on shared space, and durability. For example network policy declarations change slowly and have stringent durability requirements. Conversely, logic using link load information relies on rapidly-changing network state that is more transient in nature (and thus does not have the same durability requirements).

Second, distinct applications often have different requirements for the consistency of the network state they manage. Link state information and network policy configurations are extreme examples: transiently-inconsistent status flags of adjacent links are easier for an application to resolve than an inconsistency in network-wide policy declaration. In the latter case, a human may be needed to perform the resolution correctly.

Onix supports an application's ability to choose between update speeds and durability by providing two separate mechanisms for distributing network state updates between Onix instances: one designed for high update rates with guaranteed availability, and one designed with durability and consistency in mind. Following the example of many distributed storage systems that allow applications to make performance/scalability tradeoffs [2, 8, 29, 31], Onix makes application designers responsible for explicitly determining their preferred mechanism for any given state in the NIB – they can also opt to use the NIB solely as storage for local state. Furthermore, Onix can support arbitrary storage systems if applications write their own *import* and *export modules*, which transfer data into the NIB from storage systems and out of the NIB to storage systems respectively.

In solving the applications' preference for differing consistency requirements, Onix relies on their help: it expects the applications to use the provided coordination facilities [19] to implement distributed locking or consensus protocols as needed. The platform also expects the applications to provide the implementation for handling any inconsistencies arising between updates, if they are not using strict data consistency. While applications are given the responsibility to implement the inconsistency handling, Onix provides a programmatic framework to assist the applications in doing so.

Thus, application designers are free to determine the trade-off between potentially simplified application architectures (promoting consistency and durability) and more efficient operations (with the cost of increased complexity). We now discuss the state distribution between Onix instances in more detail, as well as how Onix integrates network elements and their state into these distribution mechanisms.

4.2 State Distribution Between Onix Instances

Onix uses different mechanisms to keep state consistent between Onix instances and between Onix and the network forwarding elements. The reasons for this are twofold. First, switches generally have low-powered management CPUs and limited RAM. Therefore, the protocol should be lightweight and primarily for consistency of forwarding state. Conversely, Onix instances can run on high powered general compute platforms and don't have such limitations. Secondly, the requirements for managing switch state are much narrower and better defined than that needed by any given application.

Onix implements a transactional persistent database backed by a replicated state machine for disseminating all state updates requiring durability and simplified consistency management. The replicated database comes with severe performance limitations, and therefore it is intended to serve only as a reliable dissemination mechanism for slowly changing network state. The transactional database provides a flexible SQL-based querying API together with triggers and rich data models for applications to use directly, as necessary.

To integrate the replicated database with the NIB, Onix includes import/export modules that interact with the database. These components load and store entity declarations and their attributes from/to the transactional database. Applications can easily group NIB modifications together into a single transaction to be exported to the database. When the import module receives a trigger invocation from the database about changed database contents, it applies the changes to the NIB.

For network state needing high update rates and availability, Onix provides a one-hop, eventually-consistent, memory-only DHT (similar to Dynamo [9]), relaxing the consistency and durability guarantees provided by the replicated database. In addition to the common

get/put API, the DHT provides soft-state triggers: the application can register to receive a callback when a particular value gets updated, after which the trigger must be reinstalled. False positives are allowed to simplify the implementation of the DHT replication mechanism. The DHT implementation manages its membership state and assigns key-range responsibilities using the same coordination mechanisms provided to applications.

Updates to the DHT by multiple Onix instances can lead to state inconsistencies. For instance, while using triggers, the application must be carefully prepared for any race conditions that could occur due to multiple writers and callback delays. Also, the introduction of a second storage system may result in inconsistencies in the NIB. In such cases, the Onix DHT returns multiple values for a given key, and it is up to the applications to provide conflict resolution, or avoid these conditions by using distributed coordination mechanisms.

4.3 Network Element State Management

The Onix design does not dictate a particular protocol for managing network element forwarding state. Rather, the primary interface to the application is the NIB, and any suitable protocol supported by the elements in the network can be used under the covers to keep the NIB entities in sync with the actual network state. In this section we describe the network element state management protocols currently supported by Onix.

OpenFlow [23] provides a performance-optimized channel to the switches for managing forwarding tables and quickly learning port status changes (which may have an impact on reachability within the network). Onix turns OpenFlow events and operations into state that it stores in the NIB entities. For instance, when an application adds a flow entry to a `ForwardingTable` entity in the NIB, the OpenFlow export component will translate that into an OpenFlow operation that adds the entry to the switch TCAM. Similarly, the TCAM entries are accessible to the application in the contents of the `ForwardingTable` entity.

For managing and accessing general switch configuration and status information, an Onix instance can opt to connect to a switch over a configuration database protocol (such as the one supported by Open vSwitch [26]). Typically this database interface exposes the switch internals that OpenFlow does not. For Onix, the protocol provides a mechanism to receive a stream of switch state updates, as well as to push changes to the switch state. The low-level semantics of the protocol closely resembles the transactional database (used between controllers) discussed above, but instead of requiring full SQL support from the switches, the database interface has a more restricted query language that does not provide joins.

Similarly to the integration with OpenFlow, Onix provides convenient, data-oriented access to the switch configuration state by mapping the switch database contents to NIB entities that can be read and modified by the applications. For example, by creating and attaching `Port` entities with proper attributes to a `ForwardingEngine` entity (which corresponds to a single switch datapath), applications can configure new tunnel endpoints without knowing that this translates to an update transaction sent to the corresponding switch.

4.4 Consistency and Coordination

The NIB is the central integration point for multiple data sources (other Onix instances as well as connected network elements); that is, the state distribution mechanisms do not interact directly with each other, but rather they import and export state into the NIB. To support multiple applications with possibly very different scalability and reliability requirements, Onix requires the applications to declare what data should be imported to and exported from a particular source. Applications do this through the configuration of import and export modules.

The NIB integrates the data sources without requiring strong consistency, and as a result, the state updates to be imported into NIB may be inconsistent either due to the inconsistency of state within an individual data source (DHT) or due to inconsistencies between data sources. To this end, Onix expects the applications to register inconsistency resolution logic with the platform. Applications have two means to do so. First, in Onix, entities are C++ classes that the application may extend, and thus, applications are expected simply to use inheritance to embed *referential inconsistency detection logic* into entities so that applications are not exposed to inconsistent state.[8] Second, the plugins the applications pass to the import/export components implement *conflict resolution logic*, allowing the import modules to know how to resolve situations where both the local NIB and the data source have changes for the same state.

For example, consider a new `Node` N, imported into the NIB from the replicated database. If N contains a reference in its list of ports to `Port` P that has not yet been imported (because they are retrieved from the network elements, not from the replicated database), the application might prefer that N not expose a reference to P to the control logic until P has been imported. Furthermore, if the application is using the DHT to store statistics about the number of packets forwarded by N, it is possible for the import module of an Onix instance to retrieve two different values for this number from the DHT (*e.g.*, due to rebalancing of controllers' responsibilities within a cluster, resulting in two controllers transiently updating the same value). The

[8] Any inconsistent changes remain pending within the NIB until they can be applied or applications deem it invalid for good.

application's conflict resolution logic must reconcile these values, storing only one into the NIB and back out to the DHT.

This leaves the application with a consistent topology data model. However, the application still needs to react to Onix instance failures and use the provided coordination mechanisms to determine which instances are responsible for different portions of the NIB. As these responsibilities shift within the cluster, the application must instruct the corresponding import and export modules to adjust their behaviors.

For coordination, Onix embeds Zookeeper [19] and provides applications with an object-oriented API to its filesystem-like hierarchical namespace, convenient for realizing distributed algorithms for consensus, group membership, and failure detection. While some applications may prefer to use Zookeeper's services directly to store persistent configuration state instead of the transactional database, for most the object size limitations of Zookeeper and convenience of accessing the configuration state directly through the NIB are a reason to favor the transactional database.

5 Implementation

Onix consists of roughly 150,000 lines of C++ and integrates a number of third party libraries. At its simplest, Onix is a harness which contains logic for communicating with the network elements, aggregating that information into the NIB, and providing a framework in which application programmers can write a management application.

A single Onix instance can run across multiple processes, each implemented using a different programming language, if necessary. Processes are interconnected using the same RPC system that Onix instances can use among themselves, but instead of running over TCP/IP it runs over local IPC connections. In this model, supporting a new programming language becomes a matter of writing a few thousand lines of integration code, typically in the new language itself. Onix currently supports C++, Python, and Java.

Independent of the programming language, all software modules in Onix are written as loosely-coupled components, which can be replaced with others without recompiling Onix as long as the component's binary interface remains the same. Components can be loaded and unloaded dynamically and designers can express dependencies between components to ensure they are loaded and unloaded in the proper order.

6 Applications

In this section, we discuss some applications currently being built on top of Onix. In keeping with the focus of the paper, we limit the applications discussed to those that are being developed for production environments. We believe the range of functionality they cover demonstrates the generality of the platform. Table 2 lists the ways in which these applications stress the various Onix features.

Ethane. For enterprise networks, we have built a network management application similar to Ethane [5] to enforce network security policies. Using the Flow-based Management Language (FML) [18] network administrators can declare security policies in a centralized fashion using high-level names instead of network-level addresses and identifiers. The application processes the first packet of every flow obtained from the first hop switch: it tracks hosts' current locations, applies the security policies, and if the flow is approved, sets up the forwarding state for the flow through the network to the destination host. The link state of the network is discovered through LLDP messages sent by Onix instances as each switch connects.

Since the aggregate flow traffic of a large network can easily exceed the capacity of a single server, large-scale deployment of our implementation, it requires multiple Onix instances to partition the flow processing. Further, having Onix on the flow-setup path makes failover between multiple instances particularly important.

Partitioning the flow-processing state requires that all controllers be able to set up paths in the network, end to end. Therefore, each Onix instance needs to know the location of all end-points as well as the link state of the network. However, it is not particularly important that this information be strongly consistent between controllers. At worst, a flow is routed to an old location of the host over a failed link, which is impossible to avoid during network element failures. It is also unnecessary for the link state to be persistent, since this information is obtained dynamically. Therefore, the controllers can use the DHT for storing link-state, which allows tens of thousands of updates per second (see Section 7).

Distributed Virtual Switch (DVS). In virtualized enterprise network environments, the network edge consists of virtual, software-based L2 switch appliances within hypervisors instead of physical network switches [26]. It is not uncommon for virtual deployments (especially in cloud-hosting providers) to consist of tens of VMs per server, and to have hundreds, thousands or tens of thousands of VMs in total. These environments can also be highly dynamic, such that VMs are added, deleted and migrated on the fly.

To cope with such environments, the concept of a distributed virtual switch (DVS) has arisen [33]. A DVS roughly operates as follows. It provides a logical switch abstraction over which policies (*e.g.*, policing, QoS, ACLs) are declared over the logical switch ports. These ports are bound to virtual machines through integration with the hypervisor. As the machines come and go and move around the network, the DVS ensures that the

Control Logic	Flow Setup	Distribution	Availability	Integration
Ethane	✓		✓	
Distributed virtual switch				✓
Multi-tenant virtualized datacenter		✓		✓
Scale-out carrier-grade IP router			✓	

Table 2: Aspects of Onix especially stressed by deployed control logic applications.

policies follow the VMs and therefore do not have to be reconfigured manually; to this end, the DVS integrates to the host virtualization platform.

Thus, when operating as part of a DVS application, Onix is not involved in forwarding plane flow setup, but only invoked when VMs are created, destroyed, or migrated. Hypervisors are organized as pools consisting of a reasonably small number of hypervisors and VMs typically do not migrate across pools; and therefore, the control logic can easily partition itself according to these pools. A single Onix instance then handles all the hypervisors of a single pool. All the switch configuration state is persisted to the transactional database, whereas all VM locations are not shared between Onix instances.

If an Onix instance goes down, the network can still operate. However, VM dynamics will no longer be allowed. Therefore, high availability in such an environment is less critical than in the Ethane environment described previously, in which an Onix crash would render the network inoperable to new flows. In our DVS application, for simplicity reasons reliability is achieved through a cold standby prepared to boot in a failure condition.

Multi-tenant virtualized data centers. Multi-tenant environments exacerbate the problems described in the context of the previous application. The problem statement is similar, however: in addition to handling end-host dynamics, the network must also enforce both addressing and resource isolation between tenant networks. Tenant networks may have, for example, overlapping MAC or IP addresses, and may run over the same physical infrastructure.

We have developed an application on top of Onix which allows the creation of tenant-specific L2 networks. These networks provide a standard Ethernet service model and can be configured independently of each other and can span physical network subnets.

The control logic isolates tenant networks by encapsulating tenants' packets at the edge, before they enter the physical network, and decapsulating them when they either enter another hypervisor or are released to the Internet. For each tenant virtual network, the control logic establishes tunnels pair-wise between all the hypervisors running VMs attached to the tenant virtual network. As a result, the number of required tunnels is $O(N^2)$, and thus, with potentially tens of thousands of VMs per tenant network, the state for just tunnels may grow beyond the capacity of a single Onix instance, not to mention that the switch connections can be equally numerous.[9]

Therefore, the control logic partitions the tenant network so that multiple Onix instances share responsibility for the network. A single Onix instance manages only a subset of hypervisors, but publishes the tunnel end-point information over the DHT so any other instances needing to set up a tunnel involving one of those hypervisors can configure the DHT import module to load the relevant information into the NIB. The tunnels themselves are stateless, and thus, multiple hypervisors can send traffic to a single receiving tunnel end-point.

Scale-out carrier-grade IP router. We are currently considering a design to create a scale-out BGP router using commodity switching components as the forwarding plane. This project is still in the design phase, but we include it here to demonstrate how Onix can be used with traditional network control logic.

In our design, Onix provides the "glue" between the physical hardware (a collection of commodity switches) and the control plane (an open source BGP stack). Onix is therefore responsible for aggregating the disparate hardware devices and presenting them to the control logic as a single forwarding plane, consisting of an L2/L3 table, and a set of ports. Onix is also responsible for translating the RIB, as calculated by the BGP stack, into flow entries across the cluster of commodity switches.

In essence, Onix will provide the logic to build a scale-out chassis from the switches. The backplane of the chassis is realized through the use of multiple connections and multi-pathing between the switches, and individual switches act as line-cards. If a single switch fails, Onix will alert the routing stack that the associated ports on the full chassis have gone offline. However, this should not affect the other switches within the cluster.

The control traffic from the network (*e.g.*, BGP or IGP traffic) is forwarded from the switches to Onix, which annotates it with the correct logical switch port and forwards it to the routing stack. Because only a handful of

[9]The VMs of a single tenant are not likely to share physical servers to avoid fate-sharing in hardware failure conditions.

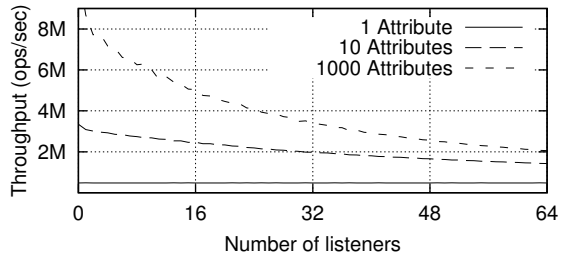


Figure 3: Attribute modification throughput as the number of listeners attached to the NIB increases.

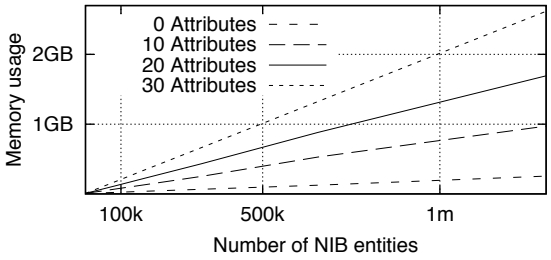


Figure 4: Memory usage as the number of NIB entities increases.

switches are used, the memory and processing demands of this applications are relatively modest. A single Onix instance with an active failover (on which the hardware configuration state is persistent) is sufficient for even very large deployments. This application is discussed in more detail in [7].

7 Evaluation

In this section, we evaluate Onix in two ways: with micro-benchmarks, designed to test Onix's performance as a general platform, and with end-to-end performance measurements of an in-development Onix application in a test environment.

7.1 Scalability Micro-Benchmarks

Single-node performance. We first benchmark three key scalability-related aspects of a single Onix instance: throughput of the NIB, memory usage of the NIB, and bandwidth in the presence of many connections.

The NIB is the focal point of the API, and the performance of an application will depend on the capacity the NIB has for processing updates and notifying listeners. To measure this throughput, we ran a micro-benchmark where an application repeatedly acquired exclusive access to the NIB (by its cooperative thread acquiring the CPU), modified integer attributes of an entity (which triggers immediate notification of any registered listeners), and then released NIB access. In this test, none of the listeners acted on the notifications of NIB changes they received. Figure 3 contains the results. With only a single attribute modification, this micro-benchmark essentially becomes

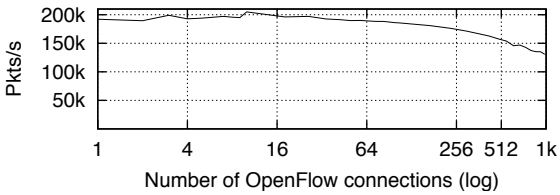


Figure 5: Number of 64-byte packets forwarded per second by a single Onix node, as the # of switch connections increases.

a benchmark for our threading library, as acquiring exclusive access to the NIB translates to a context switch. As the number of modified attributes between context switches increases, the effective throughput increases because the modifications involve only a short, fine-tuned code path through the NIB to the listeners.

Onix NIB entities provide convenient state access for the application as well as for import and export modules. The NIB must thus be able to handle a large number of entries without excessive memory usage. Figure 4 displays the results of measuring the total memory consumption of the C++ process holding the NIB while varying both network topology size and the number of attributes per entity. Each attribute in this test is 16 bytes (on average), with an 8-byte attribute identifier (plus C++ string overhead); in addition, Onix uses a map to store attributes (for indexing purposes) that reserves memory in discrete chunks. A zero-attribute entity, including the overhead of storing and indexing it in the NIB, consumes 191 bytes. The results in Figure 4 suggest a single Onix instance (on a server-grade machine) can easily handle networks of millions of entities. As entities include more attributes, their sizes increase proportionally.

Each Onix instance has to connect to the switches it manages. To stress this interface, we connected a (software) switch cloud to a single Onix instance and ran an application that, after receiving a 64-byte packet from a random switch, made a forwarding decision without updating the switch's forwarding tables. That is, the application sent the packet back to the switch with forwarding directions for that packet alone. Because of the application's simplicity, the test effectively benchmarks the performance of our OpenFlow stack, which has the same code path for both packets and network events (such as port events). Figure 5 shows the stack can perform well (forwarding over one hundred thousand packets per second), with up to roughly one thousand concurrent connections. We have not yet optimized our implementation in this regard, and the results highlight a known limitation of our threading library, which forces the OpenFlow protocol stack to do more threading context switches as the number of connections increases. Bumps in the graph are due to the operating system scheduling the controller process over multiple CPU cores.

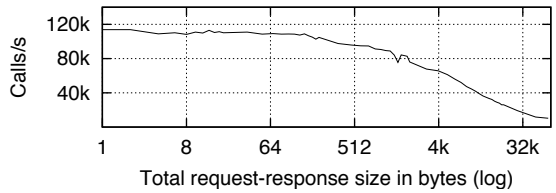


Figure 6: RPC calls per second processed by a single Onix node, as the size of the RPC request-response pair increases.

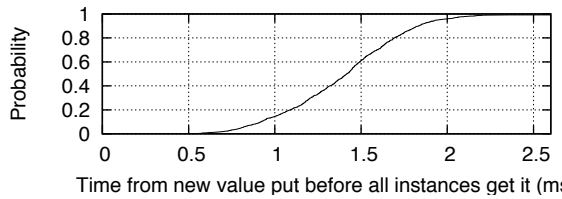


Figure 7: A CDF showing the latency of updating a DHT value at one node, and for that update to be fetched by another node in a 5-node network.

Multi-node performance. Onix instances use three mechanisms to cooperate: two state update dissemination mechanisms (the DHT and the replicated, transactional database) and the Zookeeper coordination mechanism. Zookeeper's performance has been studied elsewhere [19], so we focus on the DHT and replicated database.

The throughput of our memory-based DHT is effectively limited by the Onix RPC stack. Figure 6 shows the call throughput between an Onix instance acting as an RPC client, and another acting as an RPC server, with the client pipelining requests to compensate for network latency. The DHT performance can then be seen as the RPC performance divided by the replication factor. While a single value update may result in both a notification call and subsequent get calls from each Onix instance having an interest in the value, the high RPC throughput still shows our DHT to be capable of handling very dynamic network state. For example, if you assume that an application fully replicates the NIB to 5 Onix instances, then each NIB update will result in 22 RPC request-response pairs (2 to store two copies of the data in the DHT, $2*5$ to notify all instances of the update, and $2*5$ for all instances to fetch the new value from both replicas and reinstall their triggers). Given the results in Figure 6, this implies that the application, in aggregate, can handle 24,000 small DHT value updates per second. In a real deployment this might translate, for example, to updating a load attribute on 24,000 link entities every second – a fairly ambitious scale for any physical network that is controlled by just five Onix instances. Applications can use aggregation and NIB partitioning to scale further.

Our replicated transactional database is not optimized for throughput. However, its performance has not yet become a bottleneck due to the relatively static nature of the data it stores. Table 3 shows the throughput for different query batching sizes (1/3 of queries are INSERTs, and 2/3 are SELECTs) in a 5-node replicated database. If the application stores its port inventory in the replicated database, for example, without any batching it can process 17 port additions and removals per second, along with about 6.5 queries per second from each node about the existence of ports ($17 + 6.5 * 5 \sim 49.7$).

Queries/trans	**1**	**10**	**20**	**50**	**100**
Queries/s	49.7	331.9	520.1	541.7	494.4

Table 3: The throughput of Onix's replicated database.

7.2 Reliability Micro-Benchmarks

A primary consideration for production deployments is reliability in the face of failures. We now consider the three failure modes a control application needs to handle: link failures, switch failures, and Onix instance failures. Finally, we consider the perceived network communication failure time with an Onix application.

Link and switch failures. Onix instances monitor their connections to switches using aggressive keepalives. Similarly, switches monitor their links (and tunnels) using hardware-based probing (such as 802.1ag CFM [1]). Both of these can be fine-tuned to meet application requirements.

Once a link or switch failure is reported to the control application, the latencies involved in disseminating the failure-related state updates throughout the Onix cluster become essential; they define the absolute minimum time the control application will take to react to the failure *throughout* the network.

Figure 7 shows the latencies of DHT value propagation in a 5-node, LAN-connected network. However, once the controllers are more distant from each other in the network, the DHT's pull-based approach begins to introduce additional latencies compared to the ideal push-based methods common in distributed network protocols today. Also, the new value being put to the DHT may be placed on an Onix instance not on the physical path between the instance updating the value and the one interested in the new value. Thus, in the worst case, a state update may take four times as long as it takes to push the value (one hop to put the new value, one to notify an interested Onix instance, and two to get the new value).

In practice, however, this overhead tends not to impact network performance, because practical availability requirements for production traffic require the control application to prepare for switch and link failures proactively by using backup paths.

Onix instance failures. The application has to detect failed Onix instances and then reconfigure responsibilities within the Onix cluster. For this, applications rely on the

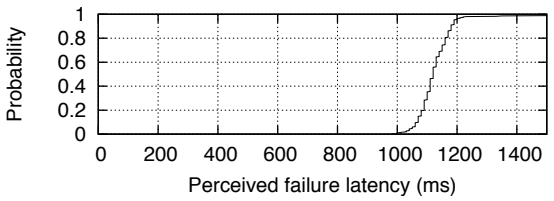


Figure 8: A CDF of the perceived communication disruption time between two hosts when an intermediate switch fails. These measurements include the one-second (application-configurable) keepalive timeout used by Onix. The hosts measure the disruption time by sending a ping every 10 ms and counting the number of missed replies.

Zookeeper coordination facilities provided by Onix. As with its throughput, we refer the reader to a previous study [19] for details.

Application test. Onix is currently being used by a number of organizations as the platform for building commercial applications. While scaling work and testing is ongoing, applications have managed networks of up to 64 switches with a single Onix instance, and Onix has been tested in clusters of up to 5 instances.

We now measure the end-to-end failure reaction time of the multi-tenant virtualized data center application (Section 6). The core of the application is a set of tunnels creating an L2 overlay. If a switch hosting a tunnel fails, the application must patch up the network quickly to ensure continued connectivity withing the overlay.

Figure 8 shows how quickly the application can create new tunnels to reestablish the connectivity between hosts when a switch hosting a tunnel fails. The measured time includes the time for Onix to detect the switch failure, and for the application to decide on a new switch to host the tunnel, create the new tunnel endpoints, and update the switch forwarding tables. The figure shows the median disruption for the host-to-host communication is 1120 ms. Given the configured one-second switch failure detection time, this suggests it takes Onix 120 ms to repair the tunnel once the failure has been detected. Although this application is unoptimized, we believe these results hold promise that a complete application on Onix can achieve reactive properties on par with traditional routing implementations.

8 Related Work

As mentioned in Section 1, Onix descends from a long line of work in which the control plane is separated from the dataplane [3–6, 15, 16, 23], but Onix's focus on being a production-quality control platform for large-scale networks led us to focus more on reliability, scalability, and generality than previous systems. Ours is not the first system to consider network control as a distributed systems problem [10, 20], although we do not anticipate the need to run our platform on end-hosts, due to the flexibility of merchant silicon and other efforts to generalize the forwarding plane [23], and the rapid increase in power of commodity servers.

An orthogonal line of research focuses on offering network developers an extensible forwarding plane (*e.g.*, RouteBricks [11], Click [22] and XORP [17]); Onix is complementary to these systems in offering an extensible control plane. Similarly, Onix can be the platform for flexible data center network architectures such as SEATTLE [21], VL2 [14] and Portland [25] to manage large data centers. This was explored somewhat in [30].

Other recent work [34] reduces the load of a centralized controller by distributing network state amongst switches. Onix focuses on the problem of providing generic distributed state management APIs at the controller, instead of focusing on a particular approach to scale. We view this work as distinct but compatible, as this technique could be implemented within Onix.

Onix also follows the path of many earlier distributed systems that rely on applications' help to relax consistency requirements in order to improve the efficiency of state replication. Bayou [31], PRACTI [2], WheelFS [29] and PNUTS [8] are examples of such systems.

9 Conclusion

The SDN paradigm uses the control platform to simplify network control implementations. Rather than forcing developers to deal directly with the details of the physical infrastructure, the control platform handles the lower-level issues and allows developers to program their control logic on a high-level API. In so doing, Onix essentially turns networking problems into distributed systems problem, resolvable by concepts and paradigms familiar for distributed systems developers.

However, this paper is not about the *ideology* of SDN, but about its *implementation*. The crucial enabler of this approach is the control platform, and in this paper we present Onix as an existence proof that such control platforms are feasible. In fact, Onix required no novel mechanisms, but instead involves only the judicious use of standard distributed system design practices.

What we should make clear, however, is that Onix does not, by itself, solve all the problems of network management. The designers of management applications still have to understand the scalability implications of their design. Onix provides general tools for managing state, but it does not magically make problems of scale and consistency disappear. We are still learning how to build control logic on the Onix API, but in the examples we have encountered so far management applications are far easier to build with Onix than without it.

Acknowledgments

We thank the OSDI reviewers, and in particular our shepherd Dave Andersen, for their helpful comments. We also thank the various team members at Google, NEC, and Nicira who provided their feedback on the design and implementation of Onix. We gratefully acknowledge Satoshi Hieda at NEC, who ran measurements that appear in this paper.

References

[1] 802.1ag - Connectivity Fault Management Standard. `http://www.ieee802.org/1/pages/802.1ag.html`.

[2] BELARAMANI, N., DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANI, A., YALAGANDULA, P., AND ZHENG, J. PRACTI Replication. In *Proc. NSDI* (May 2006).

[3] CAESAR, M., CALDWELL, D., FEAMSTER, N., REXFORD, J., SHAIKH, A., AND VAN DER MERWE, K. Design and Implementation of a Routing Control Platform. In *Proc. NSDI* (April 2005).

[4] CAI, Z., DINU, F., ZHENG, J., COX, A. L., AND NG, T. S. E. The Preliminary Design and Implementation of the Maestro Network Control Platform. Tech. rep., Rice University, Department of Computer Science, October 2008.

[5] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: Taking Control of the Enterprise. In *Proc. SIGCOMM* (August 2007).

[6] CASADO, M., GARFINKEL, T., AKELLA, A., FREEDMAN, M. J., BONEH, D., MCKEOWN, N., AND SHENKER, S. SANE: A Protection Architecture for Enterprise Networks. In *Proc. Usenix Security* (August 2006).

[7] CASADO, M., KOPONEN, T., RAMANATHAN, R., AND SHENKER, S. Virtualizing the Network Forwarding Plane. In *Proc. PRESTO* (November 2010).

[8] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proc. VLDB* (August 2008).

[9] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. SOSP* (October 2007).

[10] DIXON, C., KRISHNAMURTHY, A., AND ANDERSON, T. An End to the Middle. In *Proc. HotOS* (May 2009).

[11] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *Proc. SOSP* (October 2009).

[12] FARREL, A., VASSEUR, J.-P., AND ASH, J. A Path Computation Element (PCE)-Based Architecture, August 2006. RFC 4655.

[13] GODFREY, P. B., GANICHEV, I., SHENKER, S., AND STOICA, I. Pathlet Routing. In *Proc. SIGCOMM* (August 2009).

[14] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *Proc. SIGCOMM* (August 2009).

[15] GREENBERG, A., HJALMTYSSON, G., MALTZ, D. A., MYERS, A., REXFORD, J., XIE, G., YAN, H., ZHAN, J., AND ZHANG, H. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM CCR 35*, 5 (2005), 41–54.

[16] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. NOX: Towards an Operating System for Networks. In *SIGCOMM CCR* (July 2008).

[17] HANDLEY, M., KOHLER, E., GHOSH, A., HODSON, O., AND RADOSLAVOV, P. Designing Extensible IP Router Software. In *Proc. NSDI* (May 2005).

[18] HINRICHS, T. L., GUDE, N. S., CASADO, M., MITCHELL, J. C., AND SHENKER, S. Practical Declarative Network Management. In *Proc. of SIGCOMM WREN* (August 2009).

[19] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-Scale Systems. In *Proc. Usenix Annual Technical Conference* (June 2010).

[20] JOHN, J. P., KATZ-BASSETT, E., KRISHNAMURTHY, A., ANDERSON, T., AND VENKATARAMANI, A. Consensus Routing: The Internet as a Distributed System. In *Proc. NSDI* (April 2008).

[21] KIM, C., CAESAR, M., AND REXFORD, J. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *Proc. SIGCOMM* (August 2008).

[22] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *ACM Trans. on Computer Systems 18*, 3 (August 2000), 263–297.

[23] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR 38*, 2 (2008), 69–74.

[24] Multiprotocol Label Switching Working Group. `http://datatracker.ietf.org/wg/mpls/`.

[25] MYSORE, R. N., PAMBORIS, A., FARRINGTON, N., HUANG, N., MIRI, P., RADHAKRISHNAN, S., SUBRAM, V., AND VADHAT, A. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proc. SIGCOMM* (August 2009).

[26] PFAFF, B., PETTIT, J., KOPONEN, T., AMIDON, K., CASADO, M., AND SHENKER, S. Extending Networking into the Virtualization Layer. In *Proc. HotNets* (October 2009).

[27] Private Network-Network Interface Specification Version 1.1 (PNNI 1.1), April 2002. ATM Forum.

[28] SHERWOOD, R., GIBB, G., YAP, K.-K., APPENZELLER, G., CASADO, M., MCKEOWN, N., AND PARULKAR, G. Can the Production Network Be the Testbed? In *Proc. OSDI* (October 2010).

[29] STRIBLING, J., SOVRAN, Y., ZHANG, I., PRETZER, X., LI, J., KAASHOEK, M. F., AND MORRIS, R. Flexible, Wide-Area Storage for Distributed Systems with WheelFS. In *Proc. NSDI* (April 2009).

[30] TAVAKOLI, A., CASADO, M., KOPONEN, T., AND SHENKER, S. Applying NOX to the Datacenter. In *Proc. HotNets* (October 2009).

[31] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. SOSP* (December 1995).

[32] TOUCH, J., AND PERLMAN, R. Transparent Interconnection of Lots of Links (TRILL): Problem and Applicability Statement. RFC 5556, IETF, May 2009.

[33] VMware vNetwork Distributed Switch, Simplify Virtual Machine Networking. `http://vmware.com/products/vnetwork-distributed-switch`.

[34] YU, M., REXFORD, J., FREEDMAN, M. J., AND WANG, J. Scalable Flow-Based Networking with DIFANE. In *Proc. SIGCOMM* (August 2010).

Can the Production Network Be the Testbed?

Rob Sherwood[*], *Glen Gibb*[†], *Kok-Kiong Yap*[†], *Guido Appenzeller* [‡], *Martin Casado*[◇], *Nick McKeown*[†], *Guru Parulkar*[†]

[*] *Deutsche Telekom Inc. R&D Lab, Los Altos, CA* [†] *Stanford University, Palo Alto, CA*

[◇] *Nicira Networks, Palo Alto, CA* [‡] *Big Switch Networks, Palo Alto, CA*

Abstract

A persistent problem in computer network research is validation. When deciding how to evaluate a new feature or bug fix, a researcher or operator must trade-off realism (in terms of scale, actual user traffic, real equipment) and cost (larger scale costs more money, real user traffic likely requires downtime, and real equipment requires vendor adoption which can take years). Building a realistic testbed is hard because "real" networking takes place on closed, commercial switches and routers with special purpose hardware. But if we build our testbed from software switches, they run several orders of magnitude slower. Even if we build a realistic network testbed, it is hard to scale, because it is special purpose and is in addition to the regular network. It needs its own location, support and dedicated links. For a testbed to have global reach takes investment beyond the reach of most researchers.

In this paper, we describe a way to build a testbed that is embedded in—and thus grows with—the network. The technique—embodied in our first prototype, FlowVisor—slices the network hardware by placing a layer between the control plane and the data plane. We demonstrate that FlowVisor slices our own production network, with legacy protocols running in their own protected slice, alongside experiments created by researchers. The basic idea is that if unmodified hardware supports some basic primitives (in our prototype, OpenFlow, but others are possible), then a worldwide testbed can ride on the coat-tails of deployments, at no extra expense. Further, we evaluate the performance impact and describe how FlowVisor is deployed at seven other campuses as part of a wider evaluation platform.

1 Introduction

For many years the networking research community has grappled with how best to evaluate new research ideas.

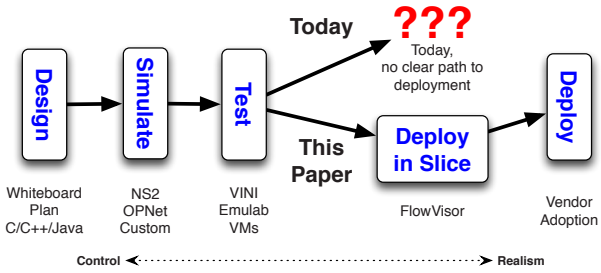


Figure 1: Today's evaluation process is a continuum from controlled but synthetic to uncontrolled but realistic testing, with no clear path to vendor adoption.

Simulation [17, 19] and emulation [25] provide tightly controlled environments to run repeatable experiments, but lack scale and realism; they neither extend all the way to the end-user nor carry real user traffic. Special isolated testbeds [10, 22, 3] allow testing at scale, and can carry real user traffic, but are usually dedicated to a particular type of experiment and are beyond the budget of most researchers.

Without the means to realistically test a new idea there has been relatively little technology transfer from the research lab to real-world networks. Network vendors are understandably reluctant to incorporate new features before they have been thoroughly tested at scale, in realistic conditions with real user traffic. This slows the pace of innovation, and many good ideas never see the light of day.

Peeking over the wall to the distributed systems community, things are much better. PlanetLab has proved invaluable as a way to test new distributed applications at scale (over 1,000 nodes worldwide), realistically (it runs real services, and real users opt in), and offers a straightforward path to real deployment (services developed in a PlanetLab slice are easily ported to dedicated servers).

In the past few years, the networking research community has sought an equivalent platform, funded by pro-

grams such as GENI [8], FIRE [6], etc. The goal is to allow new network algorithms, features, protocols or services to be deployed at scale, with real user traffic, on a real topology, at line-rate, with real users; and in a manner that the prototype service can easily be transferred to run in a production network. Examples of experimental new services might include a new routing protocol, a network load-balancer, novel methods for data center routing, access control, novel hand-off schemes for mobile users or mobile virtual machines, network energy managers, and so on.

The network testbeds that come closest to achieving this today are VINI [1] and Emulab [25]: both provide a shared physical infrastructure allowing multiple simultaneous experiments to evaluate new services on a physical testbed. Users may develop code to modify both the data plane and the control plane within their own isolated topology. Experiments may run real routing software, and expose their experiments to real network events. Emulab is concentrated in one location, whereas VINI is spread out across a wide area network.

VINI and Emulab trade off realism for flexibility in three main ways.

Speed: In both testbeds packet processing and forwarding is done in software by a conventional CPU. This makes it easy to program a new service, but means it runs much slower than in a real network. Real networks in enterprises, data centers, college campuses and backbones are built from switches and routers based on ASICs. ASICs consistently outperform CPU-based devices in terms of data-rate, cost and power; for example, a single switching chip today can process over 600Gb/s [2].

Scale: Because VINI and Emulab don't run new networking protocols on real hardware, they must always exist as a parallel testbed, which limits their scale. It would, for example, be prohibitively expensive to build a VINI or Emulab testbed to evaluate data-center-scale experiments requiring thousands or tens of thousands of switches, each with a capacity of hundreds of gigabits per second. VINI's geographic scope is limited by the locations willing to host special servers (42 today). Without enormous investment, it is unlikely to grow to global scale. Emulab can grow larger, as it is housed under one roof, but is still unlikely to grow to a size representative of a large network.

Technology transfer: An experiment running on a network of CPUs takes considerable effort to transfer to specialized hardware; the development styles are quite different, and the development cycle of hardware takes many years and requires many millions of dollars.

But perhaps the biggest limitation of a dedicated testbed is that it requires special infrastructure: equipment has to be developed, deployed, maintained and supported; and when the equipment is obsolete it needs to be updated. Networking testbeds rarely last more than one generation of technology, and so the immense engineering effort is quickly lost.

Our goal is to solve this problem. We set out to answer the following question: can we build a testbed that is embedded into every switch and router of the production network (in college campuses, data centers, WANs, enterprises, WiFi networks, and so on), so that the testbed would automatically scale with the global network, riding on its coat-tails *with no additional hardware*? If this were possible, then our college campus networks—for example—interconnected as they are by worldwide backbones, could be used simultaneously for production traffic and new WAN routing experiments; similarly, an existing data center with thousands of switches can be used to try out new routing schemes. Many of the goals of programs like GENI and FIRE could be met without needing dedicated network infrastructure.

In this paper, we introduce FlowVisor which aims to turn the production network itself into a testbed (Figure 1). That is, FlowVisor allows experimenters to evaluate ideas directly in the production network (not running in a dedicated testbed alongside it) by "slicing" the hardware already installed. Experimenters try out their ideas in an isolated slice, without the need for dedicated servers or specialized hardware.

1.1 Contributions.

We believe our work makes five main contributions:

Runs on deployed hardware and at real line-rates. FlowVisor introduces a software *slicing layer* between the forwarding and control planes on network devices. While FlowVisor could slice any control plane message format, in practice we implement the slicing layer with OpenFlow [16]. To our knowledge, no previously proposed slicing mechanism allows a user-defined control plane to control the forwarding in deployed production hardware. Note that this would not be possible with VLANs—while they crudely separate classes of traffic, they provide no means to control the forwarding plane. We describe the slicing layer in §2 and FlowVisor's architecture in §3.

Allows real users to opt-in on a per-flow basis. FlowVisor has a policy language that maps flows to slices. By modifying this mapping, users can easily try new services, and experimenters can entice users to bring real traffic. We describe the rules for mapping flows to slices in §3.2.

Ports easily to non-sliced networks. FlowVisor (and its slicing) is transparent to both data and control planes, and therefore, the control logic is unaware of the slicing

layer. This property provides a direct path for vendor adoption. In our OpenFlow-based implementation, neither the OpenFlow switches or the controllers need be modified to interoperate with FlowVisor (§3.3).

Enforces strong *isolation* between slices. FlowVisor blocks and rewrites control messages as they cross the slicing layer. Actions of one slice are prevented from affecting another, allowing experiments to safely coexist with real production traffic. We describe the details of the isolation mechanisms in §4 and evaluate their effectiveness in §5.

Operates on deployed networks FlowVisor has been deployed in our production campus network for the last 7 months. Our deployment consists of 20+ users, 40+ network devices, a production traffic slice, and four standing experimental slices. In §6, we describe our current deployment and future plans to expand into seven other campus networks and two research backbones in the coming year.

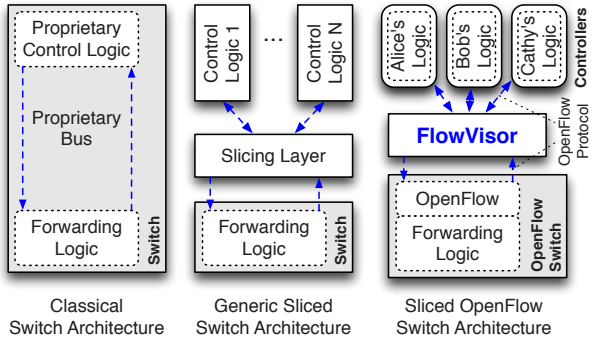


Figure 2: Classical network device architectures have distinct forwarding and control logic elements (left). By adding a transparent *slicing layer* between the forwarding and control elements, FlowVisor allows multiple control logics to manage the same forwarding element (middle). In implementation, FlowVisor uses OpenFlow and sits between an OpenFlow switch—the forwarding element—and multiple OpenFlow controllers—the control logic (right).

2 Slicing Control & Data Planes

On today's commercial switches and routers, the control plane and data planes are usually logically distinct but physically co-located. The control plane creates and populates the data plane with forwarding rules, which the data plane enforces. In a nutshell, FlowVisor assumes that the control plane can be separated from the data plane, and it then *slices* the communication between them. This slicing approach can work several ways: for example, there might already be a clean interface between the control and data planes *inside* the switch. More likely, they are separated by a common protocol (*e.g.*, OpenFlow [16] or ForCes [7]). In either case, FlowVisor sits *between* the control and data planes, and from this vantage point enables a single data plane to be controlled by multiple control planes—each belonging to a separate experiment.

With FlowVisor, each experiment runs in their own slice of the network. A researcher, Bob, begins by requesting a network slice from Alice, his network administrator. The request specifies his requirements including topology, bandwidth, and the set of traffic—defined by a set of flows, or *flowspace*—that the slice controls. Within his slice, Bob has his own control plane where he puts the control logic that defines how packets are forwarded and rewritten in his experiment. For example, imagine that Bob wants to create a new *http* load-balancer to spread port 80 traffic over multiple web servers. He requests a slice: its topology should encompass the web servers, and its flowspace should include all flows with port 80. He is allocated a control plane where he adds his load-balancing logic to control how flows are routed in the data plane. He may advertise his new service so as to attract users. Interested users "opt-in" by contacting their network administrator to add a subset of their flows to the flowspace of Bob's slice.

In this example, FlowVisor allocates a control plane for Bob, and allows him to control his flows (but no others) in the data plane. Any events associated with his flows (*e.g.* when a new flow starts) are sent to his control plane. FlowVisor enforces his slice's topology by only allowing him to control switches within his slice.

FlowVisor slices the network along multiple dimensions, including topology, bandwidth, and forwarding table entries. Slices are isolated from each other, so that actions in one slice—be they faulty, malicious, or otherwise—do not impact other slices.

2.1 Slicing OpenFlow

While architecturally FlowVisor can slice any data plane/control plane communication channel, we built our prototype on top of OpenFlow.

OpenFlow [16, 18] is an open standard that allows researchers to directly control the way packets are routed in the network. As described above, in a classical network architecture, the control logic and the data path are co-located on the same device and communicate via an internal proprietary protocol and bus. In OpenFlow, the control logic is moved to an external *controller* (typically a commodity PC); the controller talks to the datapath (over the network itself) using the OpenFlow protocol (Figure 2, right). The OpenFlow protocol abstracts

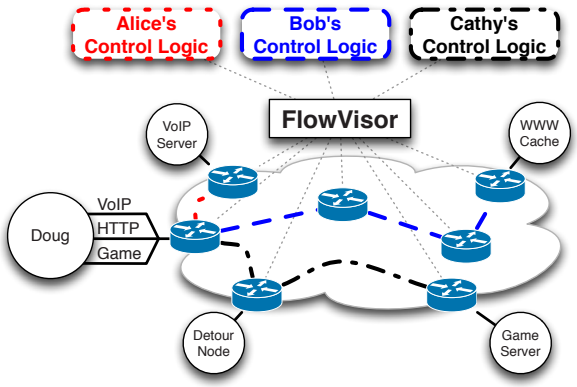


Figure 3: FlowVisor allows users (Doug) to delegate control of subsets of their traffic to distinct researchers (Alice, Bob, Cathy). Each research experiment runs in its own, isolated network slice.

forwarding/routing directives as "flow entries". A flow entry consists of a bit pattern, a list of actions, and a set of counters. Each flow entry states "perform this list of actions on all packets in this flow" where a typical action is "forward the packet out port X" and the flow is defined as the set of packets that match the given bit pattern. The collection of flow entries on a network device is called the "flow table".

When a packet arrives at a switch or router, the device looks up the packet in the flow table and performs the corresponding set of actions. If the packet doesn't match any entry, the packet is queued and a new flow event is sent across the network to the OpenFlow controller. The controller responds by adding a new rule to the flow table to handle the queued packet. Subsequent packets in the same flow will be handled without contacting the controller. Thus, the external controller need only be contacted for the first packet in a flow; subsequent packets are forwarded at the switch's full line rate.

Architecturally, OpenFlow exploits the fact that modern switches and routers already logically implement flow entries and flow tables—typically in hardware as TCAMs. As such, a network device can be made OpenFlow-compliant via firmware upgrade.

Note that while OpenFlow allows researchers to experiment with new network protocols on deployed hardware, only a single researcher can use/control an OpenFlow-enabled network at a time. As a result, without FlowVisor, OpenFlow-based research is limited to isolated testbeds, limiting its scope and realism. Thus, FlowVisor's ability to slice a production network is an orthogonal and indepenent contribution to OpenFlow-like software-defined networks.

3 FlowVisor Design

To restate our main goal, FlowVisor aims to use the production network as a testbed. In operation, the FlowVisor slices the network by slicing each of the network's corresponding packet forwarding devices (*e.g.*, switches and routers) and links (Figure 3).

With the FlowVisor,

- Network resources are sliced in terms of their bandwidth, topology, forward table entries, and device CPU (§3.1).
- Each slice has control over a set of flows, called its *flowspace*. Users can arbitrarily add (opt-in) and remove (opt-out) their own flows from a slice's flowspace at anytime (§3.2).
- Each slice has its own distinct, programmable control logic, that manages how packets are forwarded and rewritten for traffic in the slice's flowspace. In practice, each slice owner implements their slice-specific control logic as an OpenFlow controller. The FlowVisor interposes between data and control planes by proxying connections between OpenFlow switches and each slice controller (§3.3).
- Slices are defined using a slice definition policy language. The language specifies the slice's resource limits, flowspace, and controller's location in terms of IP and TCP port-pair (§3.4).

3.1 Slicing Network Resources

Slicing a network means correctly slicing all of the corresponding network resources. There are four primary slicing dimensions:

Topology. Each slice has its own view of network nodes (e.g., switches and routers) and the connectivity between them. In this way, slices can experience simulated network events such as link failure and forwarding loops.

Bandwidth. Each slice has its own fraction of bandwidth on each link. Failure to isolate bandwidth would allow one slice to affect, or even starve, another slice's throughput.

Device CPU. Each slice is limited to what fraction of each device's CPU that it can consume. Switches and routers typically have very limited general purpose computational resources. Without proper CPU slicing, switches will stop forwarding slow-path packets (§5.3.2), drop statistics requests, and, most importantly, will stop processing updates to the forwarding table.

Forwarding Tables. Each slice has a finite quota of forwarding rules. Network devices typically support a finite

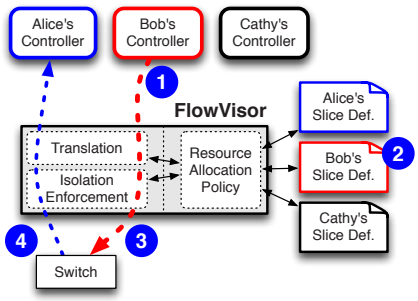


Figure 4: The FlowVisor intercepts OpenFlow messages from guest controllers (1) and, using the user's slicing policy (2), transparently rewrites (3) the message to control only a slice of the network. Messages from switches (4) are forwarded only to guests if it matches their slice policy.

number of forwarding rules (e.g., TCAM entries). Failure to isolate forwarding entries between slices might allow one slice to prevent another from forwarding packets.

3.2 Flowspace and Opt-In

A slice controls a subset of traffic in the network. The subset is defined by a collection of packet headers that form a well-defined (but not necessarily contiguous) subspace of the entire space of possible packet headers. Abstractly, if packet headers have n bits, then the set of all possible packet header forms an n-dimensional space. An arriving packet is a single point in that space representing all packets with the same header. Similar to the geometric representation used to describe access control lists for packet classification [14], we use this abstraction to partition the space into regions (flowspace) and map those regions to slices.

The flowspace abstraction helps us manage users who opt-in. To opt-in to a new experiment or service, users signal to the network administrator that they would like to add a subset of their flows to a slice's flowspace. Users can precisely decide their level of involvement in an experiment. For example, one user might opt-in all of their traffic to a single experiment, while another user might just opt-in traffic for one application (e.g., port 80 for HTTP), or even just a specific flow (by exactly specifying all of the fields of a header). In our prototype the opt-in process is manual; but in a ideal system, the user would be authenticated and their request checked automatically against a policy.

For the purposes of testbed we concluded flow-level opt-in is adequate—in fact, it seems quite powerful. Another approach might be to opt-in individual packets, which would be more onerous.

3.3 Control Message Slicing

By design, FlowVisor is a slicing layer interposed between data and control planes of each device in the network. In implementation, FlowVisor acts as a transparent proxy between OpenFlow-enabled network devices (acting as dumb data planes) and multiple OpenFlow *slice* controllers (acting as programmable control logic—Figure 4). All OpenFlow messages between the switch and the controller are sent through FlowVisor. FlowVisor uses the OpenFlow protocol to communicate upwards to the slice controllers and and downwards to OpenFlow switches. Because FlowVisor is transparent, the slice controllers require no modification and believe they are communicating directly with the switches.

We illustrate the FlowVisor's operation by extending the example from §2 (Figure 4). Recall that a researcher, Bob, has created a slice that is an HTTP proxy designed to spread all HTTP traffic over a set of web servers. While the controller will work on any HTTP traffic, Bob's FlowVisor policy slices the network so that he only sees traffic from users that have opted-in to his slice. His slice controller doesn't know the network has been sliced, so doesn't realize it only sees a subset of the HTTP traffic. The slice controller thinks it can control, i.e., insert flow entries for, all HTTP traffic from any user. When Bob's controller sends a flow entry to the switches (*e.g.*, to redirect HTTP traffic to a particular server), FlowVisor intercepts it (Figure 4-1), examines Bob's slice policy (Figure 4-2), and rewrites the entry to include only traffic from the allowed source (Figure 4-3). Hence the controller is controlling only the flows it is allowed to, without knowing that the FlowVisor is slicing the network underneath. Similarly, messages that are sourced from the switch (*e.g.*, a new flow event—Figure 4-4) are only forwarded to guest controllers whose flowspace match the message. That is, it will only be forwarded to Bob if the new flow is HTTP traffic from a user that has opted-in to his slice.

Thus, FlowVisor enforces transparency and isolation between slices by inspecting, rewriting, and policing OpenFlow messages as they pass. Depending on the resource allocation policy, message type, destination, and content, the FlowVisor will forward a given message unchanged, translate it to a suitable message and forward, or "bounce" the message back to its sender in the form of an OpenFlow error message. For a message sent from slice controller to switch, FlowVisor ensures that the message acts only on traffic within the resources assigned to the slice. For a message in the opposite direction (switch to controller), the FlowVisor examines the message content to infer the corresponding slice(s) to which the message should be forwarded. Slice controllers only receive messages that are relevant to their

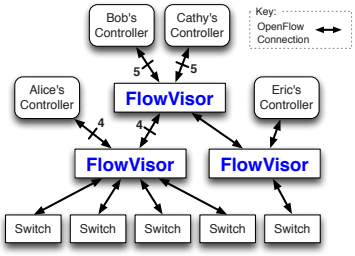


Figure 5: FlowVisor can trivially recursively slice an already sliced network, creating hierarchies of FlowVisors.

network slice. Thus, from a slice controller's perspective, FlowVisor appears as a switch (or a network of switches); from a switch's perspective, FlowVisor appears as a controller.

FlowVisor does not require a 1-to-1 mapping between FlowVisor instances and physical switches. One FlowVisor instance can slice multiple physical switches, and even re-slice an already sliced network (Figure 5) .

3.4 Slice Definition Policy

The slice policy defines the network resources, flowspace, and OpenFlow slice controller allocated to each slice. Each policy is described by a text configuration file—one file per slice. In terms of resources, the policy defines the fraction of total link bandwidth available to this slice (§4.3) and the budget for switch CPU and forwarding table entries. Network topology is specified as a list of network nodes and ports.

The flowspace for each slice is defined by an ordered list of tuples similar to firewall rules. Each rule description has an associated action, e.g., *allow*, *read-only*, or *deny*, and is parsed in the specified order, acting on the first matching rule. The rules define the flowspace a slice controls. Read-only rules allow slices to receive OpenFlow control messages and query switch statistics, but not to write entries into the forwarding table. Rules are allowed to overlap, as described in the example below.

Let's take a look at an example set of rules. Alice, the network administrator, wants to allow Bob to conduct an HTTP load-balancing experiment. Bob has convinced some of his colleagues to opt-in to his experiment. Alice wants to maintain control of all traffic that is not part of Bob's experiment. She wants to passively monitor all network performance, to keep an eye on Bob and the production network.

Here is a set of rules Alice could install in the FlowVisor:

Bob's Experimental Network includes all HTTP traffic to/from users who opted into his experiment. Thus, his network is described by one rule per user:

`Allow: tcp_port:80 and ip=`*`user_ip`*. OpenFlow messages from the switch matching any of these rules are forwarded to Bob's controller. Any flow entries that Bob tries to insert are modified to meet these rules.

Alice's Production Network is the complement of Bob's network. For each user in Bob's experiment, the production traffic network has a negative rule of the form: `Deny: tcp_port:80 and ip=`*`user_ip`*. The production network would have a final rule that matches all flows: `Allow: all`.

Thus, only OpenFlow messages that do not go to Bob's network are sent to the production network controller. The production controller is allowed to insert forwarding entries so long as they do not match Bob's traffic.

Alice's Monitoring Network is allowed to see all traffic in all slices. It has one rule, `Read-only: all`.

This rule-based policy, though simple, suffices for the experiments and deployment described in this paper. We expect that future FlowVisor deployments will have more specialized policy needs, and that researchers will create new resource allocation policies.

4 FlowVisor Implementation

We implemented FlowVisor in approximately 8000 lines of C and the code is publicly available for download from `www.openflow.org`. The notable parts of the implementation are the transparency and isolation mechanisms. Critical to its design, FlowVisor acts as a *transparent* slicing layer and enforces *isolation* between slices. In this section, we describe how FlowVisor rewrites control messages—both down to the forwarding plane and up to the control plane—to ensure both transparency and strong isolation. Because isolation mechanisms vary by resource, we describe each resource in turn: bandwidth, switch CPU, and forwarding table entries. In our deployment, we found that the switch CPU was the most constrained resource, so we devote particular care to describing its slicing mechanisms.

4.1 Messages to Control Plane

FlowVisor carefully rewrites messages from the OpenFlow switch to the slice controller to ensure transparency. First, FlowVisor only sends control plane messages to a slice controller if the source switch is actually in the slice's topology. Second, FlowVisor rewrites OpenFlow feature negotiation messages so that the slice controller only sees the physical switch ports that appear in the slice. Third, OpenFlow port up/port down messages are similarly pruned and only forwarded to the affected slices. Using these message rewriting techniques,

FlowVisor can easily simulate network events, such as link and node failures.

4.2 Messages to Forwarding Plane

In the opposite direction, FlowVisor also rewrites messages from the slice controller to the OpenFlow switch. The most important messages to the forwarding plane were insertions and deletions to the forwarding table. Recall (§2.1) that in OpenFlow, forwarding rules consist of a flow rule definition, i.e., a bit pattern, and a set of actions. To ensure both transparency and isolation, the FlowVisor rewrites both the flow definition and the set of actions so that they do not violate the slice's definition.

Given a forwarding rule modification, the FlowVisor rewrites the flow definition to intersect with the slice's flowspace. For example, Bob's flowspace gives him control over HTTP traffic for the set of users—e.g., users Doug and Eric—that have opted into his experiment. If Bob's slice controller tried to create a rule that affected all of Doug's traffic (HTTP and non-HTTP), then the FlowVisor would rewrite the rule to only affect the intersection, i.e., only Doug's HTTP traffic. If the intersection between the desired rule and the slice definition is null, e.g., Bob tried to affect traffic outside of his slice, e.g.., Doug's non-HTTP traffic, then the FlowVisor would drop the control message and return an error to Bob's controller. Because flowspaces are not necessarily contiguous, the intersection between the desired rule and the slice's flowspace may result in a single rule being expanded into multiple rules. For example, if Bob tried to affect all traffic in the system in a single rule, the FlowVisor would transparently expand the single rule in to two rules: one for each of Doug's and Eric's HTTP traffic.

FlowVisor also rewrites the lists of actions in a forwarding rule. For example, if Bob creates a rule to send out all ports, the rule is rewritten to send to just the subset of ports in Bob's slice. If Bob tries to send out a port that is not in his slice, the FlowVisor returns a "action is invalid" error (recall that from above, Bob's controller only discovers the ports that do exist in his slice, so only in error would he use a port outside his slice).

4.3 Bandwidth Isolation

Typically, even relatively modest commodity network hardware has some capability for basic bandwidth isolation [13]. The most recent versions of OpenFlow expose native bandwidth slicing capabilities in the form of per-port queues. The FlowVisor creates a per-slice queue on each port on the switch. The queue is configured for a fraction of link bandwidth, as defined in the slice definition. To enforce bandwidth isolation, the FlowVisor rewrites all slice forwarding table additions from "send out port X" to "send out queue Y on port X ", where Y is a slice-specific queue ID. Thus, all traffic from a given slice is mapped to the traffic class specified by the resource allocation policy. While any queuing discipline can be used (weighted fair queuing, deficit round robin, strict partition, etc.), in implementation, FlowVisor uses minimum bandwidth queues. That is, a slice configured for $X\%$ of bandwidth will receive at least $X\%$ and possibly more if the link is under-utilized. We choose minimum bandwidth queues to avoid issues of bandwidth fragmentation. We evaluate the effectiveness of bandwidth isolation in §5.

4.4 Device CPU Isolation

CPUs on commodity network hardware are typically low-power embedded processors and are easily overloaded. The problem is that in most hardware, a highly-loaded switch CPU will significantly disrupt the network. For example, when a CPU becomes overloaded, hardware forwarding will continue, but the switch will stop responding to OpenFlow requests, which causes the forwarding tables to enter an inconsistent state where routing loops become possible, and the network can quickly become unusable.

Many of the CPU-isolation mechanisms presented are not inherent to FlowVisor's design, but rather a workaround to deal with the existing hardware abstraction exposed by OpenFlow. A better long-term solution would be to expose the switch's existing process scheduling and rate-limiting features via the hardware abstraction. Some architectures, e.g., the HP ProCurve 5400, already use rate-limiters to enforce CPU isolation between OpenFlow and non-OpenFlow VLANs. Adding these features to OpenFlow is ongoing.

There are four main sources of load on a switch CPU: (1) generating new flow messages, (2) handling requests from controller, (3) forwarding "slow path" packets, and (4) internal state keeping. Each of these sources of load requires a different isolation mechanism.

New Flow Messages. In OpenFlow, when a packet arrives at a switch that does not match an entry in the flow table, a *new flow* message is sent to the controller. This process consumes processing resources on a switch and if message generation occurs too frequently, the CPU resources can be exhausted. To prevent starvation, the FlowVisor rate limits the new flow message arrival rate. In implementation, the FlowVisor tracks the new flow message arrival rate for each slice, and if it exceeds some threshold, the FlowVisor inserts a forwarding rule to drop the offending packets for a short period.

For example, the FlowVisor keeps a token-bucket style counter for each flow space rule ("Bob's slice gets (1)

all HTTP traffic and (2) all HTTPS traffic", i.e., two rules/counters). Each time the FlowVisor receives a new flow event, the token bucket that matches the flow gets decremented (for Bob's slice, packets that match HTTP count against token bucket #1, packets that match HTTPS count against #2). Once the bucket is emptied, the FlowVisor inserts a lowest-priority rule into the switch to drop all packets in that flowspace rule, i.e., from the example, if the token bucket corresponding to HTTPS is emptied, then the flowvisor will cause the switch to drop all HTTPS packets—without generating new flow events. The rule is set to expire in 1 second, so it is effectively a very coarse rate limiter. In practice, if a slice has control over "all traffic", this mechanism effectively blocks all new flow events from saturating the switch CPU or going to the controller, while allowing all existing flows to continue without change. We discuss the effectiveness of this technique in §5.3.2.

Controller Requests. The requests an OpenFlow controller sends to the switch, e.g., to edit the forwarding table or query statistics, consume CPU resources. For each slice, the FlowVisor limits CPU consumption by throttling the OpenFlow message rate to a maximum rate per second. Because the amount of CPU resources consumed vary by message type and by hardware implementation, it is future work to dynamically infer the cost of each OpenFlow message for each hardware platform.

Slow-Path Forwarding. Packets that traverse the "slow" path—i.e., not the "fast" dedicated hardware forwarding path—consume CPU resources. Thus, an OpenFlow rule that forwards packets via the slow path can consume arbitrary CPU resources.

This is because, in implementations, most switches only implement a subset of OpenFlow's functionality in their hardware. For example, the ASICs on most switches do not support sending one packet out exactly two ports (they support unicast and broadcast, but not in between). To emulate this behavior, the switches actually process these types of flows in their local CPUs, i.e,. on their slow path. Unfortunately, as mentioned above, these are embedded CPUs and are not as powerful as those on, for example, commodity PCs.

FlowVisor prevents slice controllers from inserting slow-path forwarding rules by rewriting them as one-time packet forwarding events, i.e., an OpenFlow "packet out" message. As a result, the slow-path packets are rate limited by the above two isolation mechanisms: new flow messages and controller request rate limiting.

Internal Bookkeeping. All network devices use CPU to update their internal counters, process events, update counters, etc. So, care must be taken to ensure that there is sufficient CPU available for the switch's bookkeeping. The FlowVisor accounts for this by ensuring that the above rate limits are tuned to leave sufficient CPU resources for the switch's internal function.

4.5 Flow Entry Isolation

The FlowVisor counts the number of flow entries used per slice and ensures that each slice does not exceed a preset limit. The FlowVisor increments a counter for each rule a guest controller inserts into the switch and then decrements the counter when a rule expires. Due to hardware limitations, certain switches will internally expand rules that match multiple input ports, so the FlowVisor needs to handle this case specially. When a guest controller exceeds its flow entry limit, any new rule insertions received a "table full" error message.

5 Evaluation

To motivate the efficiency and robustness of the design, in this section we evaluate the FlowVisor's scalability, performance, and isolation properties.

5.1 Scalability

A single FlowVisor instance scales well enough to serve our entire 40+ switch, 7 slice deployment with minimal load. As a result, we create an artificially high workload to evaluate our implementation's scaling limits. The FlowVisor's workload is characterized by the number of switches, slices, and flowspace rules per slice as well as the rate of new flow messages. We present the results for two types of workloads: one that matches what we **observe** from our deployment (1 slice, 35 rules per slice, 28 switches[1], 1.55 new flows per second per switch) and the other a **synthetic** workload (10 switches, 100 new flows per second per switch, 1 slice, 1000 rules per slice) designed to stress the system. In each graph, we fix three variables according to their workload and vary the forth.

Our evaluation measured FlowVisor's CPU utilization using a custom script. The script creates a configurable number of OpenFlow connections to the FlowVisor, and each connections simulates a switch that sends new flow messages to the FlowVisor at a prescribed rate. With each experiment, we configured the FlowVisor's number of slices and flowspace rules per slice. The new flow messages were carefully crafted to match only the last rule of each slice, causing the worst case behavior in the FlowVisor's linear search of the flowspace rules. Each test was run for 5 minutes and we recorded the CPU utilization of the FlowVisor process once per second, so each result is the average of 300 samples (shown

[1]This particular measurement did not include all of the switches in out network.

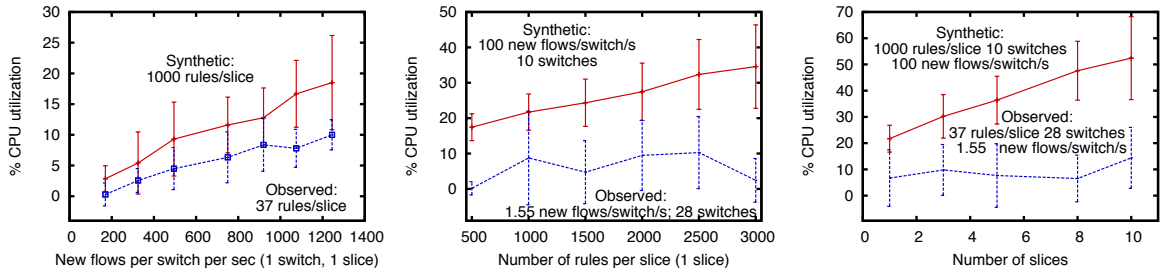


Figure 6: FlowVisor scales linearly with new flow rate, number of slices, switches, and flowspace rules. We generate high synthetic workloads to explore the scalability because the workloads observed in our deployment were non-taxing.

with one standard deviation). The FlowVisor ran on a quad-core Intel Xeon 3GHz system running 32-bit Debian Linux 5.0 (Lenny).

Our results with the synthetically high workload show that the FlowVisor's CPU load scales linearly in each of these four workload dimensions (as summarized in Figure 6). The result is promising, but not surprising. Intuitively, the FlowVisor can process a fixed number of OpenFlow messages per second (the product of number of switches by new flow rate) and each message must be matched against each rule of each slice, so the total load is approximately the product of the four workload variables. The synthetic workload with 1,000 new flows/s (10 switches by 100 new flows/s) is comparable to the peak rate of published real-world enterprise networks [20]: an 8,000 host network generated a peak rate of 1,200 new flows per second. Thus, we believe that a single FlowVisor instance could manage a large enterprise network. By contrast, our observed workload fluctuated between 0% and 10% CPU, roughly independent of the experimental variable. This validates our belief that our deployment can grow significantly using a single FlowVisor instance.

While our results show that FlowVisor scales well beyond our current requirements and workload, it is worth noting that it is possible to achieve even further scaling by moving to a multi-threaded implementation (the current implementation is single threaded) or even to multiple FlowVisor instances.

5.2 Performance Overhead

Adding an additional layer between control and data planes adds overhead to the system. However, as a result of our design, the FlowVisor does not add overhead to the data plane. That is, with FlowVisor, packets are forwarded at full line rate. Nor does the FlowVisor add overhead to the control plane: control-level calculations like route selection proceed at their un-sliced rate. FlowVisor only adds overhead to actions that cross between the control and data plane layers.

To quantify this cross-layer overhead, we measure the increased response time for slice controller requests with and without the FlowVisor. Specifically, we consider the response time of the OpenFlow messages most commonly used in our network and by our monitoring software: the new flow and the port status request messages.

In OpenFlow, a switch sends a new flow message to its controller when an arriving packet does not match any existing forwarding rules. We examine the increased delay of the new flow message to better understand how the FlowVisor affects connection setup latency. In our experiment, we connect a machine with two interfaces to a switch. One interface sends a packet every 20ms (50 packets per second) to the switch and the other interface is the OpenFlow control channel. We measure the time between sending the packet and receiving the new flow message using *libpcap*. Our results (Figure 7(a)) show that the FlowVisor increases time from the switch to controller by an average of 16ms. For latency sensitive applications, e.g., web services in large data centers, 16ms may be too much overhead. However, new flow messages add 12ms latency on average even without FlowVisor, so we believe that slice controllers in those environments will likely proactively insert flow entries into switches, avoiding this latency all together. We point out that the algorithm FlowVisor uses to process new flow messages is naive, and its run-time grows linearly with the number of flowspace rules (§5.1). We are yet to experiment with the many classification algorithms that can be expected to improve the lookup speed.

A port status request is a message sent by the controller to the switch to query the byte and packet counters for a specific port. The switch returns the counters in a corresponding port status reply message. We choose to study the port status request because we believe it to be a worst case for FlowVisor overhead. The message is very cheap to process at the switch and controller, but expensive for the FlowVisor to process: it has to edit the

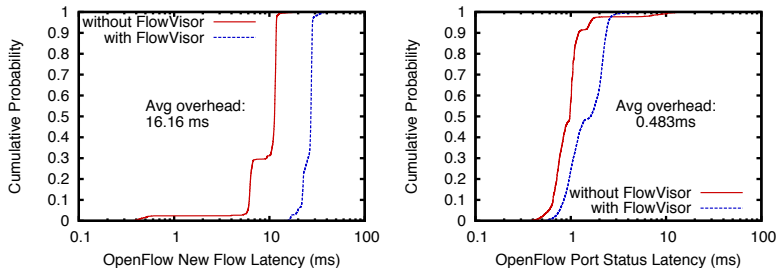


Figure 7: CDF of slicing overhead for OpenFlow new flow messages and port status requests.

message per slice to remove statistics for ports that do not appear in a sliced topology.

We wrote a special-purpose controller that sent approximately 200 port status requests per second and measured the response times. The rate was chosen to approximate the maximum request rate supported by the hardware. The controller, switch, and FlowVisor were all on the same local area network, but controller and FlowVisor were hosted on separate PCs. Obviously, the overhead can be increased by moving the FlowVisor arbitrarily far away from the controller, but we design this experiment to quantify the FlowVisor's processing overhead. Our results show that adding the FlowVisor causes an average overhead for port status responses of 0.48 milliseconds(Figure 7(b)). We believe that port status response time being faster than new flow processing time is not inherent, but simply a matter of better optimization for port status request handling.

5.3 Isolation

5.3.1 Bandwidth

To validate the FlowVisor's bandwidth isolation properties, we run an experiment where two slices compete for bandwidth on a shared link. We consider the worst case for bandwidth isolation: the first slice sends TCP-friendly traffic and the other slice sends TCP-unfriendly constant-bit-rate (CBR) traffic at full link speed (1Gbps). We believe these traffic patterns are representative of a scenario where production slice (TCP) shares a link with, for example, a slice running a DDoS experiment (CBR).

This experiment uses 3 machines—two sources and a common sink—all connected via the same HP ProCurve 5400 switch, i.e., the switch found in our wiring closet. The traffic is generated by iperf in TCP mode for the TCP traffic and UDP mode at 1Gbps for the CBR traffic. We repeat the experiment twice: with and without the FlowVisor's bandwidth isolation features enabled (Figure 8(a)). With the bandwidth isolation disabled ("without Slicing"), the CBR traffic consumes nearly all the bandwidth and the TCP traffic averages 1.2% of the link bandwidth. With the traffic isolation features enabled ("with 30/70% reservation"), the FlowVisor maps the TCP slice to a QoS class that guarantees at least 70% of link bandwidth and maps the CBR slice to a class that guarantees at least 30%. Note that theses are *minimum* bandwidth guarantees, not maximum. With the bandwidth isolation features enabled, the TCP slice achieves an average of 64.2% of the total bandwidth and the CBR an average of 28.5%. Note that the event at 20 seconds where the CBR with QoS jumps and the TCP with QoS experiences a corresponding dip. We believe this to be the result of a TCP congestion event that allowed the CBR traffic to temporarily take advantage of additional available bandwidth, exactly as the minimum bandwidth queue is designed.

5.3.2 Switch CPU

To quantify our ability to isolate the switch CPU resource, we show two experiments that monitor CPU-usage over time of a switch with and without isolation enabled. In the first experiment (Figure 8(b)), the OpenFlow controller maliciously sends port stats request messages (as above) at increasing speeds ($2, 4, 8 \ldots 1024$ requests per second). In our second experiment (Figure 8(c)), the switch generates new flow messages faster than its CPU can handle and a faulty controller does not add a new rule to match them. In both experiments, we show the switch's CPU utilization averaged over one second, and the FlowVisor's isolation features reduce the switch utilization from 100% to a configurable amount. In the first experiment, we note that the switch could handle less than 256 port status requests without appreciable CPU load, but immediately goes to 100% load when the request rate hits 256 requests per second. In the second experiment, the bursts of CPU activity in Figure 8(c) is a direct result of using null forwarding rules (§4.4) to rate limit incoming new flow messages. We expect that future versions of OpenFlow will better expose the hardware CPU limiting features already in switches today.

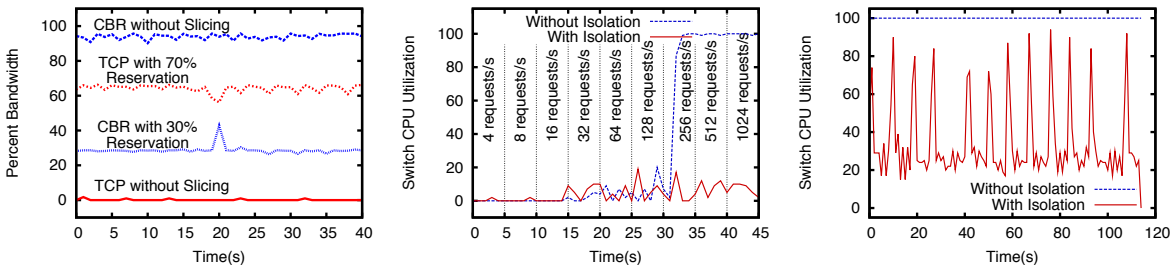


Figure 8: FlowVisor's bandwidth isolation prevents CBR traffic from starving TCP, and message throttling and new flow message rate limiting prevents CPU starvation.

6 Deployment Experience

To provide evidence that sliced experimental traffic can indeed co-exist with production traffic, we deployed FlowVisor on our production network. By "production", we refer to the network that the authors rely on to read their daily email, surf the web, etc. Additionally, six other campuses are currently using the FlowVisor as part of the GENI "meso-scale" infrastructure. In this section, we describe our experiences in deploying FlowVisor in our production network, its deployment in other campuses, and briefly describe the experiments that have run on the FlowVisor.

6.1 Stanford Deployment

At Stanford University, we have been running FlowVisor continuously on our production network since June 4th, 2009. Our network consists of 25+ users, 5 NEC IP8800 switches, 2 HP ProCurve 5400s, 30 wireless access points, 5 NetFPGA [15] cards acting as OpenFlow switches, and a WiMAX base station. Our physical network is effectively doubly sliced: first by VLANs and then by the FlowVisor. Our network trunks over 10 VLANs , including traffic for other research groups, but only three of those VLANs are OpenFlow-enabled. Of the three OpenFlow VLANs, two are sliced by FlowVisor. We maintain multiple OpenFlow VLANs and FlowVisor instances to allow FlowVisor development without impacting production traffic.

For each FlowVisor-sliced VLAN, all network devices point to a single FlowVisor instance, running on a 3.0GHz quad-core Intel Xeon with 2 GB of DRAM. For maximum uptime, we ran FlowVisor from a wrapper script that instantly restarts it if it should crash. The FlowVisor was able to handle restarts seamlessly because it does not maintain any hard state in the network. In our production slice, we ran NOX's routing module to perform basic forwarding in the network. We will publish our slicing administration tools and debugging techniques.

6.2 Deploying on Other Networks

As part of the GENI "meso-scale" project, we also deployed FlowVisor onto test networks on six university campuses, including University of Washington, Wisconsin University, Princeton University, Indiana University, Clemson University, Rutgers University. In each network, we have a staged deployment plan with the eventual goal of extending the existing OpenFlow and FlowVisor test network to their production networks. Each network runs its own FlowVisor. Recently, at the 8th GENI Engingering Conference (GEC), we demonstrated how slices at each campus's network could be combined with tunnels to create a single wide-area network slice. Currently, we are in the process of extending the FlowVisor deployment into two backbone networks (Internet2 and National Lambda Rail), with the eventual goal of creating a large-scale end-to-end sliceable wide area network.

6.3 Slicing Experience

In our experience, the two largest causes of network instability were unexpected interactions with other deployed network devices and device CPU exhaustion. One problem we had was interacting with a virtual IP feature of the router in our building. This feature allows multiple physical interfaces to act as a single, logical interface for redundancy. In implementation, the router would reply to ARP requests with the MAC address of the logical interface but source packets from any of three different MAC addresses corresponding to the physical interfaces. As a result, we had to revise the flowspace assigned to the production slice to include all four MAC addresses. Another aspect that we did not anticipate is the amount of broadcast traffic emitted from non-OpenFlow devices. It is quite common for a device to periodically send broadcast LLDP, Spanning Tree, and other packets. The level of broadcast traffic on the network made debugging more difficult and could cause loops if our OpenFlow-based loop detection/spanning tree algo-

rithms did not match the non-OpenFlow-bases spanning tree.

Another issue was the interaction between OpenFlow and device CPU usage. As discussed earlier (§ 4.4), the most frequent form of slice isolation violations occurred with device CPU. The main form of isolation violation occurred when one slice would insert a forwarding rule that could only be handled via the switch's slow path and, as a result, would push the CPU utilization to 100%, preventing slices from updating their forwarding rules. We also found that the cost to process an OpenFlow message varied significantly by type and by OpenFlow implementation particularly with stats requests, e.g., the OpenFlow aggregate stats command consumed more CPU than an OpenFlow port stats command, but not on all implementations. As part of our future work, we plan to compute a per-message type costs to each OpenFlow request to more precisely slice device CPU. Additionally, the upcoming OpenFlow version 1.1 will add support for rate limiting messages coming from the fast to slow paths.

6.4 Experiments

We've demonstrated that FlowVisor supports a wide variety of network experiments. On our production network, we ran four networking experiments, each in its own slice. All four experiments, including a network load-balancer [12], wireless streaming video [26], traffic engineering, and a hardware prototyping experiment [9], were built on top of NOX [11]. As part of the 7th GENI Engingeering Conference, each of the seven campuses demonstrated their own, locally designed experiments, running in a FlowVisor-enabled slice of the network. Our hope is that the FlowVisor will continue to allow researchers to run novel experiments in their own networks.

7 Related Work

There is a vast array of work related to network experimentation in both controlled and operational environments. Here we scratch the surface by discussing some of the more recent highlights.

The community has benefited from a number of testbeds for performing large-scale experiments. The two most widely used are PlanetLab [21] and Emulab [25]. PlanetLab's primary function has been that of an overlay testbed, hosting software services on nodes deployed around the globe. Emulab is targeted more at localized and controlled experiments run from arbitrary switch-level topologies connected by PCs. ShadowNet [3] exposes virtualization features of specific high-end routers, but does not provide per-flow forwarding control or user opt-in. VINI [1], a testbed closely affiliated with PlanetLab, further provides the ability for multiple researchers to construct arbitrary topologies of software routers while sharing the same physical infrastructure. Similarly, software virtual routers offer both programmability, reconfigurability, and have been shown to manage impressive throughput on commodity hardware (*e.g.* [5]).

In the spirit of these and other testbed technologies, FlowVisor is designed to aid research by allowing multiple projects to operate simultaneously, and in isolation, in realistic network environments. What distinguishes our approach is that we slice the hardware forwarding paths of unmodified commercial network gear.

Supercharged PlanetLab [23] is a network experimentation platform designed around CPUs and NPUs (network processors). NPUs can provide high performance and isolation while allowing for sophisticated per-packet processing. In contrast, our work forgoes the ability to perform arbitrary per-packet computation in order to work on unmodified hardware.

VLANs [4] are widely used for segmentation and isolation in networks. VLANs slice Ethernet L2 broadcast domains by decoupling virtual links from physical ports. This allows multiple virtual links to be multiplexed over a single virtual port (*trunk mode*), and it allows a single switch to be segmented into multiple, L2 broadcast networks. VLANs use a specific control logic (L2 forwarding and learning over a spanning tree). FlowVisor, on the other hand, allows users to define their own control logic. It also supports a more flexible method for defining the traffic that is in a slice, and the way users opt in. For example, with FlowVisor a user could opt-in to two different slices, whereas with VLANs their traffic would all be allocated to a single slice at Layer 2.

Perhaps the most similar to FlowVisor is the Prospero ATM Switch Divider Controller [24]. Prospero uses a hardware abstraction interface, Ariel, to allow multiple control planes to operate on the same data plane. While architecturally similar to our design, Prospero slices individual ATM switches where FlowVisor has a centralized view and can thus create a slice of the entire network. Further, Ariel provides the ability to match on ATM-related fields (e.g., VCI/VPI) where OpenFlow can match on any combination of 12-fields spanning layers one through four. This additional capability is critical for our notion of flow-level opt-in.

8 Trade-offs and Caveats

The FlowVisor *approach* is extremely general—it simply states that if we can insert a slicing layer between the control and data planes of switches and routers, then we can perform experiments in the production network. In

principle, the experimenter can exploit any capability of the data plane, so long as it is made available to them.

Our prototype of FlowVisor is based on OpenFlow, which makes very *few* of the hardware capabilities available—which limits the flexibility. Most switch and router hardware can do a lot more than is exposed via OpenFlow (*e.g.* dozens of different packet scheduling policies, encapsulation into VLANs, VPNs, GRE tunnels, MPLS, and so on). OpenFlow makes a trade-off: it only exposes a lowest common denominator that is present in all switches in return for a common vendor-agnostic interface. So far, this minimal set has met the needs of early experimenters—there appears to be a basic set of primitive "plumbing" actions that are sufficient for a wide array of experiments, and over time we would expect the OpenFlow specification to evolve to be "just enough", like the RISC instruction set in CPUs. In addition to the diverse set of experiments we have created, others have created experiments for data center network schemes (such as VL2 and Portland), new routing schemes, home network managers, mobility managers, and so on.

However, there will always be experimenters who need more control over individual packets. They might want to use features of the hardware not exposed by OpenFlow; or they might want full programmatic control, not available in any commercial hardware. The first case is a little easier to handle, because a switch or router manufacturer can expose more features to the experimenter if they choose, either by vendor-specific extensions to OpenFlow and FlowVisor, or by allowing flows to be sent to a logical internal port that, in turn, processes the packets in a pre-defined box-specific way.[2]

But if an experiment needs a way to modify packets *arbitrarily*, the researcher needs a different box. If the experiment calls for arbitrary processing in novel ways at *every* switch in the network, then OpenFlow is probably not the right interface, and our prototype is unlikely to be of much use. If the experiment only needs processing at some parts of the network (*e.g.* to do deep packet inspection, or payload processing) then the researcher can route their flows through some number of special middleboxes or way-points. The middle-boxes could be conventional servers, NPUs [23], programmable hardware [15], or custom hardware. The good thing is that these boxes can be placed anywhere, and the flows belonging to a slice can be routed through them - including all the flows from users who opt in. In the end, the value of FlowVisor to the researcher will depend on how many middle-boxes the experiment needs to be realistic—just a few and it may be worth it; if it needs hundreds or thousands then FlowVisor is providing very little value.

[2]For example, this is how some OpenFlow switches implement VPN tunnels today.

A second limitation of our prototype is the ability to create arbitrary topologies. If a physical switch is to appear multiple times in a slice's topology (*i.e.* to create a virtual topology larger than the physical topology), there is currently no standardized way to do this. The hardware needs to allow packets to loop back, and pass through the switch multiple times. In fact, most—but not all—switch hardware allows this. At some later date we expect this will be exposed via OpenFlow, but in the meantime it remains a limitation.

9 Conclusion

Put bluntly, the problem with testbeds is that they are *testbeds*. If we could test new ideas at scale, with real users, traffic and topologies, without building a testbed, then life would be much simpler. Clearly this isn't the case today: testbeds need to be built, maintained, and are expensive to deploy at scale. They become obsolete quickly, and many university machine rooms have outdated testbed equipment lying around unused.

By definition, a testbed is not the real network: therefore, we try to embed testbeds into the network by slicing the hardware. This paper described our first attempt towards embedding a testbed in the network. While not yet bullet-proof, we believe that our approach of slicing the communication between the control and data planes shows promise. Our current implementation is limited to controlling the abstraction of the forwarding element exposed by OpenFlow. We believe that exposing more fine-grained control of the forwarding elements will allow us to solve the remaining isolation issues (e.g., device cpu)—ideally with the help of the broader community. If we *can* perfect isolation, then several good things happen: researchers could validate their ideas at scale and with greater realism, the industry could perform safer quality assurance of new products, and finally, network operators could run multiple versions of the networks in parallel, allowing them to roll back to known good states.

Acknowledgments

We would like to thank Jennifer Rexford, Srinivasan Seetharaman, our shepherd Randy Katz, and the anonymous reviewers for their helpful comments and insight.

References

[1] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In vini veritas: realistic and controlled network experimentation. In *SIGCOMM '06*, pages 3–14, New York, NY, USA, 2006. ACM.

[2] BCM88130 - 630-Gbps High-Performance Packet Switch Fabric. http://www.broadcom.com/products/Switching/Carrier-and-Service-Provider/BCM88130.

[3] X. Chen, Z. M. Mao, and J. V. der Merwe. Shadownet: A platform for rapid and safe network evolution. In *Proceedings of USENIX Anual Technical Conference (USENIX'09)*. USENIX, 2009.

[4] L. S. Committee. Ieee802.1q - ieee standard for local and metropolitan area networksvirtual bridged local area networks. IEEE Computer Society, 2005.

[5] N. Egi, M. Hoerdt, L. Mathy, F. H. Adam Greenhalgh, and M. Handley. Towards High Performance Virtual Routers on Commodity Hardware. In *ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, December 2008.

[6] FIRE - Future Internet Research & Experimentation. http://cordis.europa.eu/fp7/ict/fire/.

[7] IETF ForCes. http://www.ietf.org/dyn/wg/charter/forces-charter.html.

[8] GENI.net Global Environment for Network Innovations. http://www.geni.net.

[9] G. Gibb, D. Underhill, A. Covington, T. Yabe, and N. McKeown. OpenPipes: Prototyping high-speed networking systems. In *"Proc. ACM SIGCOMM Conference (Demo)"*, Barcelona, Spain, 2009.

[10] The gigabit testbed initiative. http://www.cnri.reston.va.us/gigafr/.

[11] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards and operating system for networks. In *ACM SIGCOMM Computer Communication Review*, July 2008.

[12] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow. In *ACM SIGCOMM Demo*, August 2009.

[13] Advanced traffic management guide. HP ProCurve Switch Software Manual, March 2009. www.procurve.com.

[14] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of the ACM SIGCOMM '98*, pages 203–214, New York, NY, USA, 1998. ACM.

[15] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. Netfpga–an open platform for gigabit-rate network switching and routing. In *MSE '07: Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, pages 160–161, 2007.

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, April 2008.

[17] Ns2 network simulator. http://www.isi.edu/nsnam/ns/.

[18] The OpenFlow Switch Consortium. http://www.openflowswitch.org.

[19] Opnet technologies. http://www.opnet.com/.

[20] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A first look at modern enterprise traffic. In *ACM Internet Measurement Conference*, 2005.

[21] An open platform for developing, deploying, and accessing planetary-scale services. http://www.planet-lab.org/.

[22] J. Touch and S. Hotz. The x-bone. In *Proc. Global Internet Mini-Conference / Globecom*, 1998.

[23] J. S. Turner and et al. Supercharging planetlab: a high performance, multi-application, overlay network platform. In *SIGCOMM '07*, pages 85–96, New York, NY, USA, 2007. ACM.

[24] J. E. van der Merwe and I. M. Leslie. Switchlets and dynamic virtual atm networks. In *Proceedings of the fifth IFIP/IEEE international symposium on Integrated network management V : integrated management in a virtual world*, pages 355–368, London, UK, UK, 1997. Chapman & Hall, Ltd.

[25] B. White and J. L. et al. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[26] K.-K. Yap, T.-Y. Huang, M. Kobayashi, M. Chan, R. Sherwood, G. Parulkar, and N. McKeown. Lossless Handover with n-casting between WiFi-WiMAX on OpenRoads. In *ACM Mobicom (Demo)*, 2009.

Building Extensible Networks with Rule-Based Forwarding

Lucian Popa[*†] Norbert Egi[‡] Sylvia Ratnasamy[§] Ion Stoica[*]

Abstract

We present a network design that provides flexible and policy-compliant forwarding. Our proposal centers around a new architectural concept: that of packet *rules*. A rule is a simple if-then-else construct that describes the manner in which the network should – or should not – forward packets. A packet identifies the rule by which it is to be forwarded and routers forward each packet in accordance with its associated rule. Each packet rule is certified, guaranteeing that all parties involved in forwarding a packet agree with the packet's rule. Packets containing uncertified rules are simply dropped in the network. We present the design, implementation and evaluation of a Rule-Based Forwarding (RBF) architecture. We demonstrate flexibility by illustrating how RBF supports a variety of use cases including content caching, middlebox selection and DDoS protection. Using our prototype router implementation we show that the overhead RBF imposes is within the capabilities of modern network equipment.

1 Introduction

A central component of a network design is its forwarding architecture that determines the manner in which packets are transported between two endpoints. Today's Internet offers users a simple forwarding model: a user hands the network a packet with a destination address and the network makes a best-effort attempt to deliver the packet to the destination. Although simple, this architecture is also fairly limited and there have been repeated calls to extend the Internet's forwarding architecture for greater *flexibility*—allowing, for example, the user to select the path his packets should traverse [20, 44, 47, 49] or to specify whether packets can/should be processed by middleboxes and active routers [47, 49, 29, 48, 25].

Achieving a flexible forwarding architecture has thus been a long-held, if elusive, goal of Internet research [47, 49, 29, 20, 48, 25, 40]. Our work in this paper shares this goal. Our point of divergence from prior efforts starts with the observation that forwarding flexibility is inherently coupled with issues of *policy*.

Our thesis is that achieving flexibility is not just a matter of augmenting packets with more expressive forwarding directives that routers execute. Rather, in addition, *for each forwarding directive that enhances flexibility, the parties involved in forwarding should be able to set policies that constrain that directive*. By the policy of entity `A` (host, middlebox operator or ISP) we refer to the decision whether to approve or reject a forwarding directive based on `A`'s business or technical goals. By forwarding directive we refer to instructions provided by endpoints to routers and middleboxes on how to forward their packets. For example, a forwarding directive could specify that sender `S` can forward its packets through middlebox `M` before reaching destination `D`. An example of policy would be `M` refusing to accept packets from `S`.

To better illustrate our thesis, consider its application to the Internet. Since the main forwarding directive in IP is for sender `S` to send packets to destination `D`, `D` should be able to specify that the traffic from `S` should not reach it, *i.e.,* either by explicitly allowing or denying packets from `D`. Unfortunately, IP does not provide such functionality, effectively leaving the end-hosts vulnerable to DoS attacks. Unsurprisingly, this lack of functionality has been identified as one of the main security vulnerabilities of the Internet, and several solutions have been proposed to address this limitation [51, 52, 21, 32, 37, 22].

Of course, forwarding directives and policies are only as good as the ability of the network to enforce them and to guarantee their authenticity. What complicates policy enforcement is the involvement of multiple parties in achieving the packet's flexible behavior—the network service providers along the path, potential middlebox operators and, of course, the source and destination. As such, the network must ensure that a packet's forwarding directive complies with the policies of *all* parties involved. In our previous middlebox example, the network must ensure that `M` is willing to relay packets from `S` to `D`. If `M` does not approve, the network should simply drop the packets before reaching `M`.

In this paper, we propose a new *rule-based* forwarding architecture, RBF, that treats flexibility and policy enforcement as equal design goals. RBF is based on a new architectural concept – that of packet *rules*. In RBF, instead of sending packets to a destination (IP) address, end-hosts send packets to a rule. Rules are created by destinations. A sender fetches the destination's rule from a DNS-like infrastructure and inserts it in the packets sent to that destination.

[*]University of California, Berkeley
[†]ICSI, Berkeley
[‡]Lancaster University
[§]Intel Labs, Berkeley

A rule is a simple `if-then-else` construct that describes the manner in which the network should – or should not – forward packets. For example, a destination `A` can receive packets only from source `S` using the rule:

R_A: `if (pkt.source` $\neq$ `S) drop pkt`

Or a mobile client `B` might route certain video content through a 3rd-party transcoding proxy with:

R_B: `if (pkt.URL = hulu.com) sendPktTo trnscdPrxy`

The above examples are anecdotal (we present precise syntax and additional examples in §3) but serve to illustrate how destinations can control and customize how the network forwards their packets in a manner not easily accommodated by current IP. In effect, with rules, a receiving host must specify both *which* packets it is willing to receive as well as *how* it wants these packets forwarded and processed by the network.

The rule-based architecture we develop offers the following properties:

Rules are mandatory: routers drop packets without rules

Rules are provably authorized: all recipients (end-hosts, middleboxes and/or routers) named in the rule must explicitly agree to receive the associated packet(s). Routers, middleboxes and end-users can verify a rule's authorization.

Rules are provably safe: rules cannot exhaust network resources; *e.g.,* rules cannot compromise or corrupt routers nor cause forwarding loops.

Rules allow flexible forwarding: rules are a (constrained) program that allows a user to "customize" how the network forwards its packets.

The first two properties assist in policy enforcement by ensuring a packet is only forwarded if explicitly cleared by all recipients (*i.e.,* if it conforms with the policies of all recipients) specified in the rule. Since RBF defines policies on rules, any recipient will have the ultimate say on whether to accept any rule that contains forwarding directives sending packets to it. Since all forwarding directives are encoded into rules, we achieve our goal of enabling any entity affected by a forwarding directive to constrain that directive.

The third property ensures rules cannot be (mis)used to attack the network itself. As we shall show, the last property provides flexibility since users can give the network fine-grained instructions on how to handle their packets, enabling: explicit use of in-network functionality at middleboxes and routers, loose path forwarding, multi-path forwarding, anycast, multicast, mobility, filtering of undesired senders/ports/protocols, recording of on-path information, *etc*. In the remainder of this paper, we present the design, implementation and evaluation of a forwarding architecture that meets the above properties.

RBF relates to an extensive body of work on both forwarding flexibility and policy enforcement. We discuss related work in detail later in this paper and here only note that, at a high level, we believe what distinguishes RBF is its focus on *simultaneously* supporting flexibility and the multi-party policy requirements that such flexibility implies. As we shall see, this goal leads us to a design that differs significantly from prior proposals.

Finally, we note up-front that RBF is more complex than the existing IP forwarding architecture, which is frequently cited for its simplicity. In addition, RBF relies on strong assumptions such as anti-spoofing, the existence of rule-certifying authorities and a DNS-like infrastructure to distribute rules. The gain, relative to today's IP forwarding, is significantly improved flexibility and security; we posit that the greater complexity of our solution is a perhaps inevitable consequence of this richer service model.

2 Design Rationale and Overview

We start with the goal of network flexibility and allowing users control over how the network processes their packets. The abstraction that perhaps best supports flexibility is simply that of a *program*, leading to an architecture where users write packet-processing programs that routers execute. This vision of code-carrying packets is, of course, the cornerstone of active networking [48, 50] and we borrow this as our starting point in designing RBF. However, as we shall see, RBF severely dials back on the full-fledged generality of the original active networks' vision to arrive at a significantly simpler and safer architecture.

Rules are thus a form of program. The challenge then is to appropriately constrain these programs/rules to ensure that they cannot harm the network or other hosts. The key insight behind RBF is that these constraints must extend along *two* dimensions. First, rules must be *safe*, *i.e.,* guaranteed not to corrupt or exhaust network resources. In addition, however, we must constrain rules to respect the *policies* of all stakeholders involved—source, destination, middleboxes and ISPs. This latter requirement is unique and yet critical to networking contexts but was under appreciated in early active networking proposals.

To address policy safety, RBF incorporates two key design decisions:

(D1) Layering: we believe network operators will be unwilling to relinquish control of route discovery and computation and hence we layer RBF above current IP forwarding and do not allow rules to modify the IP-layer forwarding information base (FIB).

(D2) Verifiable stakeholder agreement: we require that a rule be authorized by all entities it explicitly names (*e.g.,* destination, middleboxes or routers). This ensures agreement of the stakeholders' policies with

the rule's intent; in particular it also ensures that rules cannot violate ISPs' routing policies, since providers must explicitly agree to have their routers named in rules. To achieve this property, in RBF rules are certified by trusted third parties, which in turn gather proofs of policy compliance from each of the rule participants.

To address rule safety, we impose strict constraints on rule syntax, such that safety can be verified through simple static analysis:

(C1) Rules cannot directly modify router state. This avoids corruption of router state. However, this can be a limiting restriction, particularly to network operators who wish to expose in-network services such as caching or monitoring to end users. To accommodate this, RBF allows operators to deploy specialized packet-processing functions at their routers and allows rules to *invoke* these functions. Such "router-defined functions" do allow rules to update router state, but only indirectly via code installed, and hence presumably trusted, by operators. This model for router-defined functions thus represents a middle ground in the tradeoff between flexibility and safety.

(C2) The rule "instruction set" is limited to only four possible *action* statements: (a) `forward` the packet to the underlying IP layer, (b) `invoke` a router-defined function, (c) `modify` the packet header and (d) `drop` the packet, plus conditionals that determine whether an action should be taken based on reading packet headers and router state. Note that there is no action that allows backward jumps across rule statements. This prevents looping or resource exhaustion at routers and ensures execution time is linear in program size.

The above constraints represent a stark departure from the rich generality of the active networks vision. Indeed, rules are more a sequence of packet steering directives, rather than a full-fledged program. The benefit is *verifiable rule and policy safety*. Moreover we find that, despite these constraints, rules suffice to express a wide variety of forwarding behaviors as we will later illustrate.

2.1 Architecture Overview and Assumptions

We now provide a brief overview of the main components and assumptions of an RBF architecture. Figure 1 illustrates the forwarding architecture of an RBF-enabled router. On receiving a packet, the router hands it to the *rule forwarding engine*, which processes the packet's rule. Such processing may involve reading router state that the network operator has opted to expose; we term such state *router attributes*. Based on information in the packet header (*packet attributes*) and router attributes, the rule forwarding engine may update the packet's attributes (including its destination), invoke router functions, drop the packet and/or hand the packet to the underlying IP forwarding engine. Recall that for safety reasons the rule is not allowed to update router state.

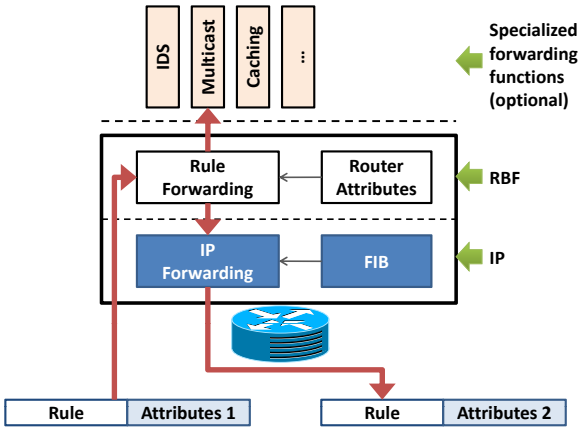


Figure 1: RBF router and rule forwarding

The design of a rule-based architecture involves the design of rules themselves as well as the surrounding infrastructure required to support the distribution, processing and securing of rules. Consequently, the RBF architecture consists of four main components:

- The specification of packet *rules* – their syntax, packet encoding, constraints on what rules can and cannot do.
- Certificate authorities called *Rule Certification Entities* (RCEs) that certify rules after checking that they are well formed, and that every destination specified in the rule agrees with (*i.e.,* has signed) the rule.
- *Modified IP routers* that verify rule certificates and process packets as described above.
- A *modified DNS infrastructure* that either directly resolves a host D's domain name to D's rule, or resolves D's domain name to another rule resolution server which in turn provides D's rule.

Assumptions: RBF builds on three major assumptions.

First, RBF assumes the existence of an anti-spoofing mechanism. This is required because rules may use source and destination IP addresses in their decision process and hence addresses must be legitimate, otherwise policy compliance cannot be enforced.[1] In this paper we assume the use of ingress filtering, although RBF can accommodate alternate solutions, *e.g.,* Passports [36]. The rationale behind our choice of ingress filtering is described in §4.

Second, we assume routers know the public keys of RCEs and can thus verify rule certificates. We assume the number of RCE organizations is relatively small and these keys can be statically configured at routers, akin to

[1]Note that any solution for blocking undesired traffic inside the network requires a way to identify sources. Anti-spoofing identifies users based on their addresses. An alternative, is to identify users by their access path [51, 52], but this approach ties communications to a specific path restricting flexibility (*e.g.,* for mobility, traffic engineering, multi-path forwarding).

how browsers today are configured with the list of major certificate authorities.[2] Note that although we assume a small number of RCE organizations, we envisage each organization will run geographically replicated instances of their service for improved scalability and robustness.

Finally, we assume that the rule resolution infrastructure (whether DNS or the resolution servers the DNS points to) is well provisioned, akin to how major Internet services (Google, DNS, Amazon) operate today, relying on engineering approaches such as maintaining a presence at major ISPs, IP anycasting, bandwidth provisioning, and so forth. As described in §4, we make this assumption to protect against "denial of rule" attacks.

Clearly, these assumptions are significant and may impede an immediate deployment of RBF in practice. And even with these assumptions, the resulting RBF design is far from trivial (for this reason, we in fact offload some of the details to an extended technical report [42]). However, we hope through the design presented in this paper to start a focused discussion about how best to practically introduce flexibility and security into the Internet and about what set of primitives routers must support to achieve this goal. In this paper we present one solution to this problem; in §10 we succinctly discuss the arguments that have led us to these specific assumptions and design.

3 The RBF Data Plane

In this section we describe the key components of the RBF data plane: rule syntax and how routers verify and execute rules. We then present examples of how rules are used.

3.1 Rule Specification

RBF represents a rule as a sequence of actions that can be conditioned by *if-then-else* instructions:

```
if (<CONDITION>) ACTION1
else ACTION2
```

Conditions are *comparison operators* applied to packet and router attributes. An action can be one of:

1. `forward` the packet to the underlying IP engine;
2. `invoke` a local function available at the router;
3. `update` the value of the packet attributes;
4. `drop` the packet.

Packet attributes include the standard IP header five-tuple (IP addresses, ports, protocol type) and, optionally, a number of custom attributes with user-defined semantics. For simplicity, RBF does not allow rules to dynamically add new attributes. Router attributes may include, for example, the router's IP address, AS number, link congestion levels, and flags indicating whether the router implements a specific function (*e.g.,* a rule can check `router.local_cache` to discover whether the router maintains a local content cache). Rules are allowed to update packet attributes, but not router attributes.

Each rule has an associated *lease* that ensures the rule can only be used for a limited period of time (§4.3). Also, every rule has an identifier (ID) defined as the concatenation of a hash of the rule owner's public key and an index unique to the owner, *hash(PK_owner):index*. In Section 7 we present an optimization to reduce packet overhead and identify most rules by using a hash over their content. This optimization can be used in the common case when there is no need for multiple rules with the same identifier; for example, mobile hosts may require different rules with the same identifier (see §3.4).

The following is an example of a rule that forwards a packet to destination `D` via a waypoint router `R1`; a packet attribute `visitedR1` indicates whether the packet has already visited `R1`:

```
R_D:
  if(packet.visitedR1 == FALSE) //from src. to R1
    if(router.address != R1)
      sendto R1
    else packet.visitedR1 = TRUE //to D
  if(packet.visitedR1)
    sendto D
```

where `sendto` involves setting the IP destination address to `D` and then handing the packet to the underlying IP forwarding engine (assuming, of course, that `D` is not the local address). Rule execution terminates at a `sendto` or `drop` action; the packet is dropped if the rule does not arrive at an explicit `sendto`. Finally, rules can `invoke` local functions at the router; after the invocation the packet is returned to the forwarding layer.

3.2 Distributing Rules to Routers

To forward a packet, a router must first obtain its rule. There are two potential approaches: (1) rules are carried in packets, (2) routers use an out-of-band mechanism to obtain rules. In RBF, we choose to carry rules in packets since the second approach would require complex rule distribution and storage protocols, and would incur extra delays in communication setup (in fact this approach would likely require special "rule-less" traffic to install rules). The tradeoff is higher overhead on the data path as rules increase packet size and routers must verify each packet's certificate; our evaluation in §7 suggests this overhead is acceptable given the capabilities of modern network equipment.

A packet with source `S` and destination `D` must include a *destination rule*, `R_D`, which is the rule specified and owned by `D`. In addition, a packet may include a *return rule*; this is the rule specified and owned by `S` and is used for return traffic from `D` to `S`.

[2] Some may regard this model of security unsatisfactory, we discuss alternatives to this deployment in §4.

3.3 Rule Verification

As mentioned earlier, rules are certified by a Rule Certification Entity (RCE) and all packets carry a signature that routers must verify. The verification load at routers is eased by two factors. First, only routers at trust boundaries need to verify rules. Second, routers can cache verification results by maintaining a hash of the rule and its signature. With caching, the full signature verification is only required for the first packet forwarded on a new rule (as long as the verification result is cached). Thus, verifications can be limited only to border routers and, assuming a large enough cache, the verification rate is given by the arrival rate of packets with new rules. By contrast, the signature length adds to the overhead of *every* packet.

Different cryptographic solutions offer different tradeoffs between signature length, signing time (incurred only at RCEs), verification time (incurred at routers) and security. Our current RBF design assumes Elliptical Curve Cryptography (ECC) because ECC signatures are shorter than RSA ones, while exhibiting similar security properties. At the same time, verification time in ECC is typically longer than RSA's. However, in practice verification can be accelerated using ASIC-based implementations or dedicated specialized co-processors. Such implementations are already commercially available [5, 7, 8] and incorporated into network appliances and routers. Furthermore, traffic measurements [4] show that new flow arrivals represent less than 1% of the link capacity on average and less than 5% of the total number of packets, a volume that can be accommodated using commercial ECC modules [5, 7] or recent research proposals [53, 34]. We evaluate different signature mechanisms briefly in § 7 and in greater detail in [42].

3.4 Examples of RBF usage

To illustrate the application of rules, we present a series of example usage scenarios; the rule syntax in these examples is largely identical to the high-level rule language supported by our RBF prototype router (§6), with simplifications for readability as appropriate.

Port-based filtering: A web server, `D`, uses the following simple rule to ensure it only receives packets on port 80:

```
R_filter_port:
  if(packet.dst_port != 80) drop;
  sendto D
```

Middlebox Support: In addition to accepting traffic directly on port 80, `D` might use the following rule to route all other incoming traffic through a packet scrubber [2, 6]. This functionality can be deployed either by `D`'s provider (as a router function), or by a third party (at a middlebox `Scrb`) as presented below:[3]

```
R_mbox_port:
  if(packet.dst_port == 80)
    sendto D                        //directly to D
  else
    if(packet.scrubbed == FALSE)    //before scrubber
      if(router.address != Scrb)
        sendto Scrb
      else                          //at scrubber
        packet.scrubbed = TRUE      //mark scrubbed
        invoke Scrb_service         //scrub
    else
      sendto D                      //after scrubber
```

Thus, similar to previous proposals [47, 49], RBF provides explicit support for middleboxes such as WAN optimizers, proxies, caches, encryption engines, transcoders, SSL offloaders, intrusion detection, *etc*.

Secure Middlebox Traversal: In the previous example, an attacker can directly send a packet with the attribute values set so as to appear that the packet has already visited the middlebox. More generally, one should be able to enforce that rule directives are respected when the rule participants (sources, middleboxes) are not trusted.

One approach to protect against this behavior is to leverage RBF's assumption that sources cannot spoof their addresses. More specifically, after each middlebox the rule can verify that the packet has indeed been sent by the required middlebox, since middleboxes/waypoints need to set the (non-spoofable) source address attribute in packets (for brevity we omit this in the presented examples); see [42] for more details on this approach.

In an alternate approach, special cryptographic functions deployed at middleboxes and destinations can be used to create/verify proofs guaranteeing the packet has visited the middlebox, as follows:

```
R_mbox_port_crypto:
  if(packet.dst_port == 80)
    sendto D                        //directly to D
  else
    if(packet.proven == FALSE)
      if(router.address != Scrb)    //before scrubber
        sendto Scrb
      else                          //at scrubber
        if(packet.scrubbed == FALSE)
          packet.scrubbed = TRUE
          invoke Scrb_service       //(1) scrub
        else                        //scrubbed
          packet.proven = TRUE
          invoke Prove              //(2) create proof
    else                            //proven
      if(router.address != D)
        sendto D
      else
        invoke VerifyAndDeliver     //check proof at D
```

In this example, the `Prove` function at the middlebox signs the immutable part of the packet header and/or payload, and adds this signature as an attribute to the packet header. In turn, the `VerifyAndDeliver` function at `D` checks the middlebox signature and, if the check succeeds, delivers the packet to the end application. Note that checking the signature requires that `D` knows the public or shared key(s) of middlebox(es); for efficiency, the middlebox could sign the hash chain of a batch of packets.

[3] Note that `Scrb` can represent the address of a load balancer used with several physical middleboxes.

DoS Protection: To protect against DDoS attacks, a server D can create a custom rule for each client that drops packets from any source other than the client. By controlling the number of rules active at a given time, D controls the maximum number of active clients (each rule has an associated lease period). An example of a rule similar to a network capability [52, 51] is:

```
R_filter_src:
  if(packet.source != requester_IP)
    drop;
  ...//rest of the rule
```

Similarly to capability based architectures [52, 51], our solution is based on the premise that destinations are able to grant rules on demand, and that any requester can ask for a destination's rule. In RBF, this task falls to the rule resolution infrastructure and raises the possibility of a "denial of rule" attack on this infrastructure (akin to denial-of-capability attacks in capability-based systems[41]). We present the details of rule resolution and discuss denial-of-rule attacks in §4.

Mobility: Host D changes its network IP address due to physical movement. In RBF, D can continue an existing communication without having to re-establish it. To achieve this, D creates a rule for the new address with the same ID as the rule used in the existing communication, and places it in the packet as the return rule.

Multicast: For security reasons, RBF does not support packet replication, and thus multicast cannot be implemented entirely at the RBF layer. Instead, multicast can be implemented by invoking multicast functionality deployed by ISPs at a subset of their routers; this functionality maintains (soft) state at routers to create a (reverse path) multicast tree. This approach implements essentially an overlay multicast solution, which leverages the IP multicast functionality at on-path routers (see [42] for details).

On-path Caching: Consider an ISP I that deploys caching functionality at some of its (border) routers. A web-service D can contract with I and use this functionality. For this purpose, D creates and publishes the following rule:

```
R_caching:
  if(router.caching_available and
          packet.crt_router != router.address)
    packet.crt_router = router.address
    invoke Caching
  sendto D
```

where the `crt_router` attribute makes sure the caching functionality is called just once at each caching-enhanced router.

In this example, the caching functionality can decide to respond to the requester directly and not forward the packets further to D, which reduces latency for the requester and traffic load at D. A similar rule can support recent proposals for content-centric routing [35, 33].

Other Examples: Our technical report [42] provides examples of applying RBF to a range of additional applications, including: secure loose path forwarding [44, 40], multipath forwarding, network diagnostics, anycast, reverse traceroute (path recording), delay-tolerant networking and even source control over middlebox or path selection. Importantly, these individual examples can be combined as needed. For example, a content distribution network can distribute load among multiple sites using anycast and, at the same time, protect its servers with on-path IDS functionality provided by ISPs.

4 The RBF Control Plane

In RBF, ISPs provide their clients with rules to access the local DNS server and a Rule Certification Entity (RCE), which can certify clients' rules. This information can be provided through a modified DHCP service, similar to the way ISPs or organizations provide the IP address of DNS servers today.

In this section, we describe the RBF mechanisms for rule creation and certification (§4.1), rule distribution (§4.2), lease enforcement (§4.3) and anti-spoofing (§4.4).

4.1 Rule Creation and Certification

To receive traffic, a client must create a rule that allows one or more sources to send traffic to it. Before distributing this rule, the client must ask an RCE to certify it. RCE certification guarantees that rules obey the policies of all stakeholders. In particular, certification guarantees the following properties:

1. Every destination in the rule (*i.e.,* any address that appears as an argument of a `sendto` instruction) has agreed to receive packets using that rule;
2. The operators providing router functions invoked by the rule approve the rule behavior;
3. The rule cannot cause infinite loops;
4. The rule cannot bypass ISP routing policies.

A client can either create rules itself and directly ask an RCE to certify these rules, or use a trusted DHCP-like service to create and certify rules on its behalf. In the remainder of this section we present the former case.

As described above, the ISP provides each client with a rule to access an RCE that has a contract with the ISP. The following example shows a possible rule that allows a client D to access an RCE named C:

$R_{D \rightarrow C}$: `if (source == D) sendto C`

Before certifying a rule, an RCE verifies that the rule has been authorized by each destination that appears in the rule. A client who has created a rule authorizes it by simply signing the rule with its private key. A client that appears in the rule as a destination, other than the rule's creator, will first verify that the content of the rule obeys

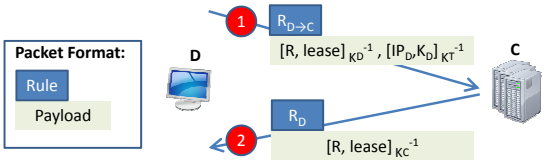


Figure 2: Rule Certification

its policies before signing the rule. For example, an intrusion detection box may verify that the destination indeed belongs to a client allowed to use the service (*e.g.*, based on a contract between the client and the provider of the intrusion detection service), a waypoint router may verify that the final destination is allowed to use source-routing, *etc*.

Let (K_D, K_D^{-1}) denote the (public, private) key pair of client D, and let $\texttt{IP}_D$ be the IP address of D. To prove to an RCE that the client signing the rule with private key K_D^{-1} indeed owns IP address $\texttt{IP}_D$, client D sends a certificate along with the signed rule that binds its public key K_D and IP address $\texttt{IP}_D$. This certificate is signed by an entity T, *i.e.*, $[IP_D, K_D]_{K_T^{-1}}$, where K_T^{-1} represents the private key of T. Clearly, the RCE must trust entity T. In fact, in our solution we will assume that T is itself an RCE.

Next, we present the rule certification process in detail, initially for the case in which the rule has a single destination, and then for the case in which the rule has multiple destinations or waypoints/middleboxes.

Certify single-destination rules: Assume destination D wishes to certify a rule R that forwards packets only to its address $\texttt{IP}_D$, *e.g.*, `R: sendto` $\texttt{IP}_D$. Also, assume D already has a rule $\texttt{R}_D$ on which it can be reached by the RCE C. D obtains this rule as part of the bootstrapping process, which we discuss later.

Fig. 2 shows the certification of D's rule, R, by C:

1. Host D signs rule R with its private key, and sends it to C using rule $\texttt{R}_{D\to C}$. In addition, D sends the certificate binding its public key and address, *i.e.*, $[IP_D, K_D]_{K_T^{-1}}$. Upon receiving this request, C verifies the certificate as well as the signature of the requested rule. These ensure that the request has been made by the owner of $\texttt{K}_D$ and that the requester is also the owner of $\texttt{IP}_D$. In addition, C verifies that R is well formed (see §5).
2. If rule verification succeeds, C signs the rule with its private key and sends it back to D using the return rule in its certification request, $\texttt{R}_D$. At this point, host D can distribute rule R to other hosts directly (as a return rule) or through DNS.

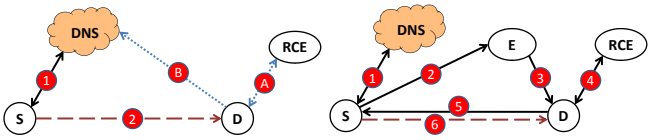
The certification procedure (Fig. 2) needs only to guarantee the authenticity of the request. Since rules are public, confidentiality is not a concern. Since the lease is an absolute value (§4.3), the only effect of replaying rule requests is increased traffic at the RCE. The maximum lease value that C can sign for a rule is negotiated between D's ISP and C. Furthermore, RCEs can limit the number of clients contacting them and can limit each user's certification rate, as we discuss in this section.

Certify multiple destination rules: In this case, every destination (*i.e.*, any host, middlebox, or waypoint router that appears as an argument of a `sendto` instruction) in a rule must agree to receive packets on that rule, *i.e.*, the rule must respect its policies. In particular, every such destination must sign the rule. One of the destinations, D, collects the signatures of all the other destinations along with their certificates binding their public keys to their addresses. D then sends this information to its RCE. In turn, the RCE verifies that all destinations in the rule have signed the rule and sends the signed rule back to D. The lease signed by the RCE has the minimum duration between the requested lease and the leases of all the certificates binding the addresses and the keys of the participants.

Certify rules invoking functions: Operators providing router functions can restrict which rules can invoke these functions. The certification process is similar to certifying multiple destination rules. The identifiers of functions whose invocation requires authorization are represented as hashes of public keys. RCEs certify a rule containing such an invocation only if the rule is signed with the private key corresponding to the function identifier.

Bootstrapping: To certify rules, client D needs to (1) know the rule to contact an RCE, C; (2) provide C with a return rule to receive the certified rule; and (3) obtain the certificate from a trusted authority that signs the binding between D's key K_D and its address IP_D. We assume the ISP provides D with a rule to access an RCE C (similarly to how ISPs today bootstrap clients' access to the DNS). Given this initial rule, we use a simple request-response exchange between the client and the RCE to obtain both the certificate binding the client's IP address to its key as well as its *first* rule. Due to space constraints, we refer the reader to our extended technical report [42] for more details on the bootstrapping process.

RCE load and availability: To control its certification load, an RCE can rate-limit the number of certification requests that it processes from each individual client. Clients are identified by IP address; the anti-spoofing mechanism prevents clients from impersonating each other. Alternatively, clients can be identified by "personalized" rules provided by the ISP to the customer to access the RCE; such rules may have a finer granularity than the anti-spoofing mechanism. RCEs can indirectly protect themselves against link-level DoS attacks by controlling the number of clients under contract.



(a) Regular Distribution (b) DDoS protection

Figure 3: Rule Distribution (solid lines = rule lookup process; dashed lines = data communication; dotted lines = setup)

RCEs must be highly available to enable rule certification at any time. RCEs can meet this requirement by using multiple servers and multiple sites. ISPs and destinations can protect themselves against RCE unavailability by contracting with multiple RCEs.

RCE Key Distribution and Revocation: In this paper we do not explore solutions for the distribution and revocation of RCE keys to routers. Here, we simply mention two possible approaches towards this goal. In one approach, RCE keys could be distributed and revoked using DNSSEC. For example, in the `txt` or other RR type, one DNS entry contains the number of RCEs and, for each RCE, there is one DNS entry (based on its index such as "ID24.rce") that contains the RCE's key. Routers periodically update the RCE keys. In another approach, RCEs could be deployed along AS boundaries, such that each AS would have its own RCE. This approach has the advantage that additional security can be enforced, *e.g.,* the trust in some RCEs can be restricted to their own address ranges. Secure BGP could be used to distribute RCE keys in this case, but at the expense of extra complexity.[4]

4.2 Rule Distribution

RBF uses an extended DNS infrastructure to distribute rules, as illustrated in Fig. 3(a). The destination `D` creates and certifies a rule for itself (step A) and inserts it into the DNS (step B). A sender `S` that wants to contact `D` looks up `D`'s name in the DNS; the DNS is extended to return `D`'s rule rather than its address (step 1). After obtaining a rule to `D`, `S` directly sends packets to `D` (step 2). Note that for practical purposes the rules of the DNS root servers need to have long leases (to avoid tedious reconfiguration or refresh protocols), as with today's long-lived addresses.

In Section 3.4 we pointed out that rules can be used to block DDoS attacks. This relies on (1) the ability to distribute customized rules to different senders (*i.e.,* give a sender `S` a rule that drops all packets not generated by `S`) and on (2) the ability to protect the rule distribution itself from DoS attacks.

To protect against DDoS attacks, client `D` can contract with a large entity `E`, and redirect its DNS entry to `E`, by registering `E`'s rule under its DNS name. Fig. 3(b) illustrates this approach. DNS will reply to a lookup for `D`'s name with `E`'s rule (step 1). The DNS entry that contains `E`'s rule must belong to a new type of DNS RR. This new class of entries is returned directly to clients by DNS resolvers. Upon a receipt of such an answer to its DNS query, the requester will continue the DNS lookup by contacting `E` (step 2). `E` rate-limits rule requests and forwards them to `D` (step 3), thus protecting `D` from DoS attacks. For the authorized requesters, `D` creates rules (step 4) and replies back to the requesters (step 5). `E` forwards requests to `D` conforming to a policy (see §3.4), which can be updated by `D` at any time.

Note that some malicious users may still get their requests forwarded by `E` and authorized by `D`. To alleviate this attack, `E` can employ fair queuing across senders, and `D` can blacklist known attackers at `E`. Such an approach offers a protection similar to network capabilities that apply per-source fair queuing at routers [37].

4.3 Rule Leases

The lease is an expiration time stamp certified along with the rule description. A router drops a packet if its current time exceeds the rule expiration time. For simplicity, in this paper we assume that all routers and RCEs are synchronized via NTP [14] as recommended by router manufacturers [19]. We present a solution that does not rely on global clock synchronization in [42].

4.4 Anti-Spoofing Mechanism

If a source can spoof addresses on packets it sends, it can send packets to a destination `D` even if the rule does not allow it to, and in this way evade `D`'s policy. Moreover, one can mount a DDoS attack by using a single rule distributed by a malicious source to a set of colluders. To address this problem, RBF can use a previously proposed anti-spoofing mechanism. In this paper, we propose the use of ingress filtering, which is already deployed by over 75% of today's ASes [23]. When deploying RBF, RBF routers could also be used to apply ingress filtering. Note that if malicious ASes do not apply ingress filtering, DoS protection is not fully compromised as only hosts in these ASes can launch attacks.

Instead of ingress filtering, RBF could leverage other anti-spoofing mechanisms such as Passport [36]. However, Passport [36] requires a secure routing layer and incurs extra overhead in packets.

The anti-spoofing mechanism requires middleboxes and routers that change a packet's destination address also to change the packet's source address attribute.

5 Security Analysis

The RBF design aims to achieve the following three goals: (i) *policy enforcement* – ensure that the authorized rules respect the policies of all participants (routers, middleboxes, destinations), and packets with unauthorized

[4]Note that DNSSEC could also potentially be used to distribute keys when RCEs are deployed along AS boundaries.

\Mechanisms Properties\	Certification	Lease	Anti-Spoofing	Static Analysis
No Rule Forging	×	×		
No Rule Tampering	×			
No Rule Evasion		×	×	
Network Safety	×			×

Table 1: Properties and Defense Mechanisms

rules are dropped inside the network; (ii) *rule enforcement* - rules cannot be used by malicous senders and, if senders or rule participants are untrusted, respect of rule directives can be enforced; and (iii) *rule safety* rules cannot be used to attack the network. Next, we summarize RBF's security properties, the threat model and assumptions under which they hold, and the mechanisms that allow RBF to meet these goals. We present a detailed analysis and proofs of RBF's security properties in [42].

Assumptions: We assume that DNS resolution is secure, that distribution of RCE keys to routers is secure, and that RCEs are not malicious.

Attackers: An attacker in RBF can be any host, middlebox, or router: sources can attempt to attack destinations by forging, evading or tampering with their rules; destinations can try to attack the network by creating rules that waste resources and slow down routers; middleboxes and routers can attempt both of these attacks.

Security Properties: We decompose the aforementioned security goals into four specific desired properties:

1. **No Rule Forging:** A host `S` cannot manufacture a rule that sends packets to another host `D`, unless `D` explicitly agrees with this rule, *i.e.,* destinations and middleboxes control the creation of rules that send traffic to them.
2. **No Rule Tampering:** Sources, routers and middleboxes cannot tamper with the destination's rules.
3. **No Rule Evasion:** Host `S` cannot send packets to destination `D`, if `D`'s rules do not accept packets from `S`.
4. **Network Safety:** A destination `D` cannot create unsafe rules. In particular, `D` cannot create rules that (a) cause infinite loops, (b) corrupt router state, (c) DoS routers or RCEs, or (d) violate ISP policies.

Mechanisms and Defenses: RBF uses four mechanisms to achieve the above properties: (1) rule certification, (2) rule leases, (3) anti-spoofing, and (4) static analysis. Table 1 summarizes which mechanisms serve to meet the four security properties.

6 Implementation

This section describes our prototype RBF router and rule compiler.

6.1 An RBF Rule Compiler

Our prototype offers users a high-level language largely identical to the syntax used in this paper in which to write rules. We wrote an RBF compiler in C++ that translates this high-level language into a compact rule format carried in packets. This compact format uses: 8B(ytes) for public-key hashes, 3B for the user-local index, 3B to identify the RCE, 3B to identify router-defined functions that do not require approval to be invoked and 8B for those that do, and 2B as the default RBF packet attribute values.[5] For the lease we use an absolute expiration time consisting of first 4B of the NTP format, with second-level granularity and a wrap-around period of 136 years. For efficiency, we use variable-length encoding in representing the internal rule structure. The maximum rule description size is 256B in our implementation.

6.2 A Prototype RBF Router

Rationale: We implemented RBF forwarding using Click [39] and RouteBricks [26]. Most commercial routers implement packet processing using ASICs or specialized network processors (NPs) rather than general-purpose CPUs and, as such, our software-based prototype is not entirely representative of currently deployed routers. To a large extent, our choice of prototyping platform is borne of necessity since commercial routers are closed. Beyond necessity, however, we believe a software-based prototype is valuable for multiple reasons. First, recent research [26, 31, 27] has demonstrated that, with modern multi-core servers, it is now possible to build high-speed software routers up to edge and even core speeds. Secondly, while not directly reusable, several aspects of our implementation architecture such as our approach to partitioning tasks across multiple cores should apply to network processor-based routers. Finally, several research [12, 28] and commercial switches [3] augment ASIC-based switches with some number of co-located general-purpose cores or servers for greater flexibility in packet processing – our prototype architecture is directly applicable to such platforms.

Design requirements: We build our prototype in the context of modern multi-core servers that incorporate multiple processors or "sockets", each with multiple cores [17, 1]. As shown in Fig. 1, the software stack of an RBF router includes the following key components: (1) an IP forwarding module, (2) the rule execution engine, and (3) some (possibly zero) number of specialized forwarding function modules. All packets traverse the rule execution and IP forwarding components, while different subsets of packets may traverse one or more specialized functions. In addition, the resources required to process a packet may vary widely across functions; *e.g.,* an en-

[5] Our current prototype only supports this default size.

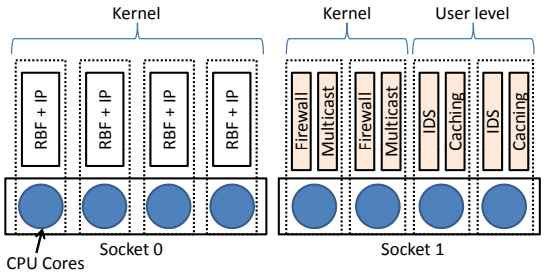


Figure 4: Core Allocation Example in RBF Router

cryption function would use lots of CPU but little cache, while a caching module may use more cache and less CPU. At a high level, our design goal is to balance high performance (*i.e.*, making efficient use of resources) with performance isolation, both across different functions, and between functions and the rule execution engine (*i.e.*, sharing resources in a fair manner).

Approach: In its full generality, the above goal requires contention-aware scheduling that simultaneously takes into account the multiple resources (cores, various caches, memory bandwidth, I/O bandwidth) for which tasks might contend. For modern multi-core systems, this is in itself an area of active research [24, 54] and beyond the scope of this paper. Instead, in our prototype, we address the issue as follows. The IP forwarding module and the rule execution engine are the central, most critical, components of the router and hence we assign these to a socket of their own and do not run specialized functions at cores in this socket. This avoids having the IP and rule execution engines contend with specialized functions for cache, CPU and other resources at the cost of some potential inefficiency since these "reserved" cores (if unused) cannot be used by specialized functions (if needed). We then assign specialized functions to the remaining "unreserved" cores. We rely on the existing (Click and Linux in our implementation) system schedulers to ensure fair sharing of CPU resources between functions on the same core.

To achieve high performance, we run a single thread performing both IP forwarding and rule execution at each of the reserved cores; this ensures that packets that do not invoke any specialized functions are processed entirely by a single core avoiding potentially expensive cache misses and inter-core synchronization [26]. Packets that invoke specialized functions must be relayed across cores and hence incur corresponding performance overheads due to cache misses and so forth. To improve the efficiency of such transfers when these functions are implemented in user space, we use shared memory pages and event queues. In our current prototype, when a rule invokes a user-level function, we make a single copy of the packet to the shared memory. An example of the resulting system architecture is depicted in Fig. 4.

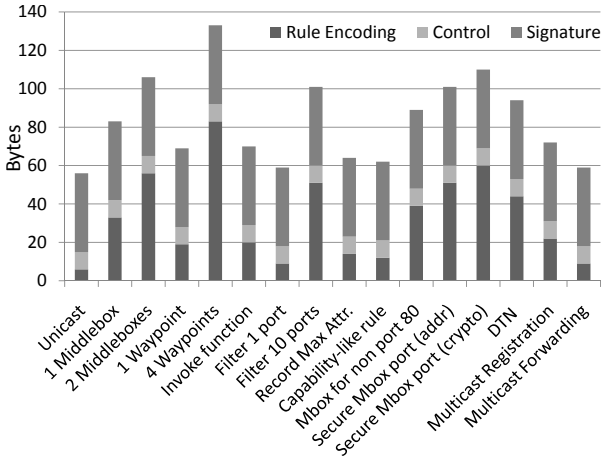


Figure 5: Rule Sizes

7 Evaluation

We use our prototype to evaluate the overhead RBF imposes on packets (§7.1), routers (§7.2) and RCEs (§7.4).

7.1 Packet Size Overhead

Fig. 5 presents rule sizes (in bytes) for a range of examples, including those from §3.4. The figure captures all the RBF-related fields and presents the size broken down into (a) the rule and the associated attributes' binary encoding; (b) the control fields used for the lease, RCE identification, to specify whether the return rule is in the packet and so forth; and (c) the rule signature. We assume a 41B signature obtained using ECDSA with ECC public keys for RCEs derived from the NIST B-163 or K-163 curves [18], offering 80 bits of security. Note that RBF is independent of the exact signature scheme used and that smaller (and faster) signatures can be used. However, shorter RCE keys may require more frequent updates to compensate for the lower security guarantees. The rules in Fig. 5 do not contain an identifier, and are identified by endpoints and routers using a hash over their content. Rule identifiers are required for rules whose content may change during a communication (such as the rules of mobile hosts) and incurs an additional 11B overhead in our implementation (8B for the hash of the public key and 3B for the user-selected index). Note that the rule identifier need be unique only with respect to a single communication endpoint (*i.e.*, all parties that a host `X` communicates with should have unique rule identifiers).

From Fig. 5 we can see that many common forwarding scenarios (unicast, routing via middleboxes, rules for DoS protection) can be expressed with around 60-80B rules while more complex rules (*e.g.*, loose source routing, secure middleboxes, anycast) can take as much as 140B. The average rule size across all examples we have implemented is 85B, representing 13% overhead for an average packet of 630B[4] and 6% overhead for a 1500B

packet. By comparison, using RSA-1024 signatures (instead of ECDSA) would incur 27% overhead on a 630B packet and 11% overhead on a 1500B packet.

Potential Optimization - Rule Caching: Per-packet overhead can be significantly reduced by *caching* rules at endpoints and routers; packets whose rules have been cached need only carry rule identifiers. There are two opportunities for caching. First, destinations can cache return rules; this allows the return rule to be eliminated from all but the first packet in a source-to-destination exchange. Second, rules can also be cached at routers. Here, however, we must ensure no packet carrying only a rule identifier arrives at a router that does not store the corresponding rule description. This might occur, for example, due to a route change or when a router deletes the rule from its cache. In such cases, the router can simply drop the packet in question, if the endpoints include the rule on all retransmissions and during periods of high packet loss. Of course, caching imposes additional storage overhead at routers as we evaluate shortly.

In summary, based on our evaluation, we see that the per-packet overhead due to RBF can range from as low as 24B when using caching and up to ∼250B in the bad case where there is no caching and the packet carries complex destination and return rules.

7.2 Router Overhead

In this section, we evaluate the overhead RBF imposes on routers for rules that do not invoke specialized processing functions; we consider router functions in the following section. The primary overhead RBF imposes on routers is the additional processing required to execute and authenticate rules and the additional storage capacity required if rules are cached. In this paper we do not evaluate rule authentication, which we assume is done by specialized hardware at trust-boundary routers; in [42] we present an evaluation for software rule authentication using RSA signatures, and show that our software router is not significantly slowed down when forwarding realistic traffic traces and performing verifications (the slowdown is less than 10%).

Rule Forwarding: We first measure the overhead of rule processing by comparing the performance of RBF-on-RouteBricks to that of unmodified RouteBricks running on a single high-end server machine. We use a dual-socket server with four 2.8GHz Intel Xeon (X5560) cores per socket to (from) which we generate (sink) traffic over two dual-port 10G NICs. In this experiment, we use all 8 cores to forward packets.

Fig. 6 plots forwarding rates for some of the examples from Fig. 5. The first column represents a packet stream with sizes generated based on a packet trace collected on the Abilene backbone [11]; since the packets from the trace do not have rules, we add to each packet the slowest

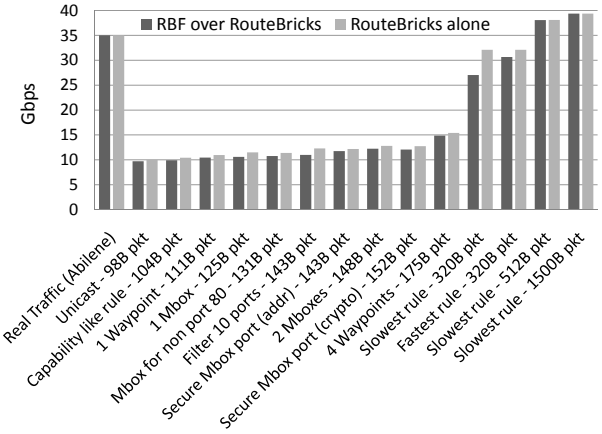
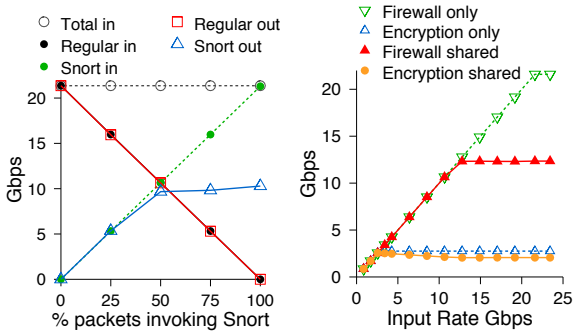


Figure 6: Forwarding speed for RBF over RouteBricks

rule that fits in the packet. By "slowest" we mean the rule that takes the longest time to forward, as determined by the number of conditions and actions encountered during forwarding. To capture the performance impact for small packets, we profile each rule without any payload and with no return rules. In the figure, packet sizes are shown next to the example name and entries are sorted in order of increasing packet size; the packet size also includes the Ethernet and IP headers. The last columns depict forwarding of larger packets, *i.e.*, that also contain data payload. To see the impact of the type of rule for these packets, we profiled them with the fastest and the slowest rules. Note that all rules are profiled in the worst case, meaning that the longest path through the rule is considered. For the slowest rule we use a 145B anycast rule which selects one out of 10 destinations based on the value of a packet attribute.

Overall, we see in Fig. 6 that the performance degradation due to RBF's more complex per-packet processing is always modest ($<$15%) and virtually non-existent at larger packet sizes. For small packets the CPU is the forwarding bottleneck, and RBF's added processing slows the router. For larger packets the I/O system is the bottleneck, and there are enough free CPU cycles to execute rules. A fine-grained profile of the rule execution module showed that it uses between 120 CPU cycles per packet for the fastest rule and 600 CPU cycles for the slowest rule; in comparison, the IP router used in our experiments requires around 3000 cycles per packet without rule execution. Also note that compared to the network-level forwarding results from Fig. 6, application-level goodput is further reduced by the RBF header.

Router cache sizes: We earlier proposed that routers cache rule authentications and/or rule descriptions. In each case, the number of cache entries required depends on the number of distinct rules the router sees. If we assume that all packets in a flow share the same rule, then the number of distinct rules passing through a given



(a) Isolation of Forwarding (b) Fairness Across Functions

Figure 7: Isolation and Fairness of Router Functions

router varies between the worst case of O(#flows) to the best case of O(#destinations) seen by the router. The former corresponds to a destination that uses a different rule for every source it communicates with, the latter to a destination that uses a single rule for all potential sources.

In our implementation, each cached authentication is 19 bytes – 11B for the rule identifier and an 8-byte hash value used to verify whether the rule has changed since it was authenticated. Each router uses its own secret hash function to prevent attackers from using hash collisions. Thus, one million rules would require only 19MB of memory. For caching entire rules, Fig. 5 reveals average and worst-case rule sizes of 85 and 133 bytes, respectively. If we conservatively assume traffic is uniformly distributed across these forwarding categories, we arrive at an estimated cache size of 85MB (average) to 133MB (worst-case) for 1M rules, which is within the scope of memory available in current routers.

7.3 Router Functions

Our router prototype supports specialized functions implemented at either kernel- or user-level. We currently support three router functions: (i) the Snort IDS [13] adapted to run as a user-level function, (ii) a kernel-mode firewall implemented in Click and (iii) a kernel-mode encryption engine also implemented in Click. Each function runs as a separate process/kernel thread isolated from the packet forwarding path through queues. We measure performance and fairness using the above functions on the same hardware as before. We dedicate four cores to the standard forwarding path and the remaining four cores to custom functions.

Fig. 7(a) illustrates the resource isolation between forwarding and router functions; the function used in this experiment is Snort (running on four cores). To generate traffic we use real traces of (moderately) malicious traffic created particularly for IDS testing [10, 30]. The average packet size of the trace used was 1065 bytes. To avoid biasing our results, we modify Snort not to drop any malicious packets so packets are only dropped due to resource exhaustion. Our test maintains constant total input traffic while increasing the percentage of input traffic that invokes Snort (X-axis). We see from Fig. 7(a) that Snort traffic does not affect the "regular" traffic that does not invoke Snort, in the sense that no regular traffic is dropped, even as a growing percentage of input Snort traffic is dropped. We observed the same isolation when using traces with small packets (see [42]).[6]

Fig. 7(b) illustrates isolation between router functions. We run three experiments: (1) all traffic invokes the firewall function and no traffic invokes encryption; (2) all traffic invokes encryption; and (3) equal halves of traffic invoking the firewall and encryption. Fig. 7(b) plots the resulting forwarding rates under increasing input traffic. In the third (shared) test the CPU is shared fairly between functions (we use Click-level scheduling); thus, the ratio between the maximum throughputs achieved by each router function is expected to roughly match the ratio between the throughputs of the functions when running in isolation. In Fig. 7(b) the encryption throughput is higher for a mix of firewalled and encrypted traffic than 50% of that when encryption is executed alone because the trace contains large packets. In this case, the CPU is not the bottleneck for the firewall functionality but is the bottleneck for encryption (since it is more CPU-intensive), and thus encryption ends up using the leftover firewall CPU cycles. If small packets are used, both functionalities achieve around 50% of their throughput in isolation [42]. Note that the high rates achieved by running each function in isolation illustrates the benefit of running instances of a single function at multiple cores (as opposed to one function per core) since this allows the unused resources from one function to be seamlessly utilized by other functions.

7.4 RCE Load

We use a simple back-of-the-envelope calculation to estimate the total number of RCE servers required for the Internet. The bulk of requests to RCEs are determined by IP address changes and per-client certifications requested by sites that protect against DoS (by redirecting DNS requests to powerful entities, see §3.4,§4.2). Note that in the latter case, requests to RCEs are made only for approved customers. There are currently around 700 million hosts in the Internet [9]; given the current trend of smart mobile devices we consider 1 billion hosts. We assume a worst-case scenario in which all hosts request certifications in the same second; these requests are made either by hosts individually to certify a rule or by

[6] We also measured the performance of the system with all the eight cores running both forwarding and Snort, and all the packets directed to Snort. While this configuration does not provide isolation for the regular traffic, it can forward a higher total throughput of 22Gbps.

	Flexibility Available to End-hosts						Security / Policy Compliance				
Property / Architecture	Explicit Middlebox Support	Multiple Paths	Invoke Router Extensions (*e.g.* IDS, multicast)	Use Router State in Forwarding (*e.g.* anycast, DTN)	Record Router State (*e.g.* network probing, ECN)	Mobility	Policy Compliant Loose Paths	Policy Compliant In-network Functionality Use	Receiver Reachability Control	Host DDoS Protection	Safety of Network & Routers (*e.g.* loops, break ISP policies)
RBF	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Active Networks	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
ESP	No	No	Yes	No	Yes	No	No	No	No	No	Yes
i3, DOA	Yes	No	No	No	No	Yes	No	No	No	No	Yes
Platypus, SNAPP	Yes	Yes	No	No	No	No	Yes	No	No	No	Yes
TVA, SIFF	No	No	No	No	No	No	No	No	No	Yes	Yes
NUTSS	Yes	No	No	No	No	No	Yes	No	Yes	No	Yes
PushBack, AITF, StopIt	No	No	No	No	No	No	No	No	No	Yes	Yes
Predicate routing, Off-by-default	No	No	No	No	No	No	No	No	Yes	No	Yes
ICING	Yes	No	No	No	No	No	Yes	No	Yes	No	Yes

Figure 8: A comparison of RBF to:Active Networks [48, 50], ESP[25], i3[47], DOA[49], Platypus[44], SNAPP[40], TVA[52], SIFF[51], NUTSS[29], PushBack[32], AITF[21], StopIt[37], Predicate Routing[45], Off-by-default[22], ICING[46]

websites hosts are trying to access. We implemented RCE rule certification in software using RSA signatures, and measured it on the same 8-core server used throughout our evaluation. We find a single server can achieve a certification rate of over 16,000 rules per second. Based on benchmarks of our implementation and assuming an oversubscription rate of 10× (ISPs today commonly oversubscribe by 100×), the total load due to certifying rules above could be accommodated by around 6,000 servers; *e.g.,* handled by 20 RCEs with 300 servers each. Hardware implementations might reduce this number by more than an order of magnitude. For example, using recent ECC prototypes [53, 34] a single ASIC could potentially perform 40,000 RCE certifications per second, requiring a total of only 2500 such devices.

8 Related Work

RBF is inspired by and extends several directions in past research. RBF's contribution is in offering extensive flexibility while respecting policies, where prior approaches tended to focus on one or the other. Fig. 8 compares the flexibility and security features of RBF with those of previous proposals. RBF is complementary to recent efforts proposing open router APIs [15, 16, 38, 12] – we offer an overall network design by which endpoints use the new functionality these router architectures promise to enable. This paper extends an earlier position paper [43] that argued the case for a rule-based architecture.

A key feature that distinguishes RBF from previous proposals and allows it to achieve both flexibility and policy compliance is its division of functionality between the data and control planes. Active Networks typically make little use of the control plane, as they deploy the forwarding functionality and enforce security on the data plane. This makes policy compliance hard to achieve. In contrast, more recent proposals such as OpenFlow [12] rely heavily on the control plane and install flow state in the network to make sure the data plane respects the appropriate policies. This approach, while simplifies the data plane, results in a more rigid architecture. For example, supporting host mobility and traffic engineering require tearing down the old paths and instantiating new ones. These are expensive operations which have a negative impact on the scalability of these proposals. In contrast, with RBF, each packet contains (in its rule) enough information to prove to routers that it respects the policies of all participants involved in forwarding the packet. RBF achieves this property despite the fact that neither the routers nor the packet contain the policies. Thus, RBF retains the datagram model of the IP, unlike other recent proposals (*e.g.,* network capabilities [52, 51], ICING [46] and OpenFlow [12]), which are more akin to a connection-oriented model. Finally, while overlay-based architectures can implement more sophisticated data plane or control plane mechanisms, they cannot leverage support at routers and are thus less powerful.[7]

9 Incremental Deployment

All the benefits of RBF shown in Fig. 8 except receiver reachability control and DDoS protection can be achieved with a partial deployment of RBF routers and middleboxes. In an initial phase, RBF routers could support both RBF and legacy (non-RBF) traffic. To also offer DoS protection and reachability control, individual ASes can upgrade to RBF by dropping legacy traffic. Hosts in such ASes can use multihoming to handle legacy traffic, although they will be vulnerable to DoS attacks on legacy interfaces.

10 Discussion

We have presented RBF, an architecture we have argued strikes a desirable balance between flexibility and the ability to guarantee policy compliance of all network entities. We started this work with two high level goals in mind. First, we wanted a *complete* architecture that supports not only previously proposed communication primitives, but also future ones. Second, we wanted an

[7]For example, overlay architectures can only drop unwanted packets at overlay nodes and hence cannot create a network that is fundamentally default-off; once the network-layer address of a node is known, it can always be attacked at the underlying network layer.

efficient architecture in which a packet unwanted by a receiver along its path is dropped as early as possible.

While completeness in this context is difficult to formalize, intuitively we have reduced it to (1) supporting arbitrary communication paths, and (2) allowing all network entities (*i.e.,* sender, receivers, middleboxes, and routers) to be involved in the decision process. In other words, we wanted to be able to define virtually any forwarding path and give all involved parties a say in defining it. We noted that such a path can be encoded by associating with each node an "if-then-else" code snippet, which specifies the next node down the path. We further noted that allowing different network entities to define the communication pattern is equivalent to allowing them to define these code snippets. This is roughly what the RBF proposal is.

These goals are ambitious – they subsume, unite and extend many years of proposals for greater flexibility and security in networks – and much of RBF's complexity follows from these goals.

Finally, one might question whether a relaxation of RBF's goals might lead to a significantly simpler design. This is a valid question that we leave for future work. We believe that understanding the fundamental tradeoffs associated with these goals is critical and, at the very least, that RBF is a step toward arriving at such an understanding.

Acknowledgments: We would like to thank our shepherd, Brad Karp, the anonymous reviewers as well as Gautam Altekar, Katerina Argyraki, Byung-Gon Chun, Mihai Dobrescu, Ali Ghodsi, Brighten Godfrey, Gianluca Iannaccone, Jayanth Kannan, Eddie Kohler, Petros Maniatis, Maziar Manesh, Sergiu Nedevschi, Costin Raiciu, Arsalan Tavakoli and Matei Zaharia for their feedback on various stages of the paper.

References

[1] AMD Opteron Server Processor. http://www.amd.com/.
[2] Arbor Networks Peakflow: www.arbornetworks.com/en/peakflow-ip-flow-based-technology.html.
[3] Arista 7100 Series Switches. http://www.aristanetworks.com/en/7100Series.
[4] CAIDA: www.caida.org/data/realtime/.
[5] Certicom Suite B IP Core, http://www.certicom.com.
[6] Cisco Traffic Anomaly Detector: www.cisco.com/en/us/products/ps5892/.
[7] CLP-17: High Performance Elliptic Curve Cryptography (ECC) Point Multiplier Core, http://www.ellipticsemi.com/products-clp-17.php.
[8] Elliptic Curve Point Multiply and Verify Core, http://www.ipcores.com/elliptic_curve_crypto_ip_core.htm.
[9] Internet Systems Consortium, Internet Host Count History, https://www.isc.org/solutions/survey/history, 2009.
[10] Lincoln laboratory: Darpa intrusion detection data set (week 6). http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/.
[11] NLANR: Internet Measurement and Analysis. http://moat.nlanr.net.
[12] Open Flow Switch Consortium. http://www.openflowswitch.org.
[13] Snort IDS/IPS, http://www.snort.org.
[14] RFC 1305 - Network Time Protocol. 1992.
[15] Juniper Networks Delivers Platform for Customer and Partner Application Development. *Juniper Press Release*, Dec. 2007.
[16] Cisco Opens Routers to Customers and Third-Party Applications. *Cisco Press Release*, April 2008.
[17] Next Generation Intel Microarchitecture (Nehalem). http://intel.com, 2008.
[18] Digital Signature Standard (DSS). *Federal Information Processing Standards Publication*, June 2009.
[19] T. Akin. Hardening Cisco Routers. *O'Reilly*, 2002.
[20] K. Argyraki and D. R. Cheriton. Loose Source Routing as a Mechanism for Traffic Policies. In *ACM SIGCOMM Workshops*, 2004.
[21] K. Argyraki and D. R. Cheriton. Active Internet Traffic Filtering: Real-time Response to Denialof-Service Attacks. In *USENIX Annual Conf.*, 2005.
[22] H. Ballani, Y. Chawathe, S. Ratnasamy, T. Roscoe, and S. Shenker. Off by Default! In *ACM HotNets*, 2005.
[23] R. Beverly and S. Bauer. The Spoofer project: Inferring the extent of source address filtering on the Internet. In *SRUTI Workshop*, 2005.
[24] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek. Reinventing Scheduling for Multicore Systems. In *HotOS*, 2009.
[25] K. L. Calvert, J. Griffioen, and S. Wen. Lightweight Network Support for Scalable End-to-End Services. In *ACM SIGCOMM*, August 2002.
[26] M. Dobrescu, N. Egi, K. Argyraki, B.-g. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *ACM SOSP*, 2009.
[27] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards High Performance Virtual Routers on Commodity Hardware. In *Proceedings of 4th Conference on Future Networking Technologies (ACM CoNEXT 2008)*, Madrid, Spain, December 2008.
[28] A. Greenhalgh, F. Huici, M. Hoerdt, P. Papadimitriou, M. Handley, and L. Mathy. Flow processing and the rise of commodity network hardware. *SIGCOMM Comput. Commun. Rev.*, 39(2), April 2009.
[29] S. Guha and P. Francis. An End-Middle-End Approach to Connection Establishment. In *ACM SIGCOMM*, 2007.
[30] J. W. Haines, R. P. Lippmann, D. J. Fried, E. Tran, S. Boswell, and M. A. Zissman. 1999 DARPA Intrusion Detection System Evaluation: Design and Procedures. In *MIT Lincoln Laboratory Technical Report*.
[31] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-Accelerated Software Router. In *ACM SIGCOMM*, 2010.
[32] J. Ioannidis and S. M. Bellovin. Implementing Pushback: Router-Based Defense Against DDoS Attacks. In *NDDS*, 2002.
[33] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking Named Content. In *ACM CoNEXT)*, 2009.
[34] K. Järvinen et al. Fast point multiplication on Koblitz curves: Parallelization method and implementations. *Microprocess. Microsyst.*, 33(2), 2009.
[35] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *ACM SIGCOMM*, 2007.
[36] X. Liu, A. Li, X. Yang, and D. Wetherall. Passport: Secure and Adoptable Source Authentication. In *USENIX NSDI*, 2008.
[37] X. Liu, X. Yang, and Y. Lu. To Filter or to Authorize: Network-Layer DoS Defense Against Multimillion-node Botnets. In *ACM SIGCOMM*, 2008.
[38] J. C. Mogul, P. Yalagandula, J. Tourrilhes, R. McGeer, S. Banerjee, T. Connors, and P. Sharma. API Design Challenges for Open Router Platforms on Proprietary Hardware. In *ACM Hotnets*, 2008.
[39] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. *SIGOPS Oper. Syst. Rev.*, 33(5):217–231, 1999.
[40] B. Parno, A. Perrig, and D. G. Andersen. SNAPP: Stateless Network-Authenticated Path Pinning. In *ACM ASIACCS*, 2008.
[41] B. Parno, D. Wendlandt, E. Shi, A. Perrig, B. Maggs, and Y.-C. Hu. Portcullis: Protecting Connection Setup from Denial-of-Capability Attacks. In *ACM SIGCOMM*, 2007.
[42] L. Popa, N. Egi, S. Ratnasamy, and I. Stoica. Rule-based Forwarding (RBF): Improving the Internet's Flexibility and Security. *UCB Technical Report*, 2010. http://www.eecs.berkeley.edu/%7Epopa/rbfTechReport.pdf.
[43] L. Popa, I. Stoica, and S. Ratnasamy. Rule-based Forwarding (RBF): improving the Internet's flexibility and security. In *ACM Hotnets*, 2009.
[44] B. Raghavan and A. C. Snoeren. A System for Authenticated Policy-Compliant Routing. In *ACM SIGCOMM*, 2004.
[45] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, and P. Jardetzky. Predicate Routing: Enabling Controlled Networking. In *ACM Hotnets*, 2002.
[46] A. Seehra, J. Nous, M. Walfish, D. Mazieres, A. Nicolosi, and S. Shenker. A Policy Framework for the Future Internet. In *ACM Hotnets*, 2009.
[47] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *ACM SIGCOMM*, 2002.
[48] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Comm.*, 1997.
[49] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *OSDI*, 2004.
[50] D. Wetherall. Active network vision and reality: Lessons from a capsulebased system. In *ACM SOSP*, 1999.
[51] A. Yaar, A. Perrig, and D. Song. SIFF: A Stateless Internet Flow Filter to Mitigate DDoS Flooding Attacks. In *IEEE Symp. on Sec. and Priv.*, 2004.
[52] X. Yang, D. J. Wetherall, and T. Anderson. A DoS-limiting Network Architecture. In *ACM SIGCOMM*, 2005.
[53] Y. Zhang, D. Chen, Y. Choi, L. Chen, and S.-B. Ko. A high performance ECC hardware implementation with instruction-level parallelism over GF(2163). *Microprocess. Microsyst.*, 34(6), 2010.
[54] S. Zhuravleva, S. Blagodurov, et al. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *ASPLOS*, 2010.

TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones

William Enck
The Pennsylvania State University

Peter Gilbert
Duke University

Byung-Gon Chun
Intel Labs

Landon P. Cox
Duke University

Jaeyeon Jung
Intel Labs

Patrick McDaniel
The Pennsylvania State University

Anmol N. Sheth
Intel Labs

Abstract

Today's smartphone operating systems frequently fail to provide users with adequate control over and visibility into how third-party applications use their private data. We address these shortcomings with TaintDroid, an efficient, system-wide dynamic taint tracking and analysis system capable of simultaneously tracking multiple sources of sensitive data. TaintDroid provides realtime analysis by leveraging Android's virtualized execution environment. TaintDroid incurs only 14% performance overhead on a CPU-bound micro-benchmark and imposes negligible overhead on interactive third-party applications. Using TaintDroid to monitor the behavior of 30 popular third-party Android applications, we found 68 instances of potential misuse of users' private information across 20 applications. Monitoring sensitive data with TaintDroid provides informed use of third-party applications for phone users and valuable input for smartphone security service firms seeking to identify misbehaving applications.

1 Introduction

A key feature of modern smartphone platforms is a centralized service for downloading third-party applications. The convenience to users and developers of such "app stores" has made mobile devices more fun and useful, and has led to an explosion of development. Apple's App Store alone served nearly 3 billion applications after only 18 months [4]. Many of these applications combine data from remote cloud services with information from local sensors such as a GPS receiver, camera, microphone, and accelerometer. Applications often have legitimate reasons for accessing this privacy sensitive data, but users would also like assurances that their data is used properly. Incidents of developers relaying private information back to the cloud [35, 12] and the privacy risks posed by seemingly innocent sensors like accelerometers [19] illustrate the danger.

Resolving the tension between the fun and utility of running third-party mobile applications and the privacy risks they pose is a critical challenge for smartphone platforms. Mobile-phone operating systems currently provide only coarse-grained controls for regulating whether an application can *access* private information, but provide little insight into how private information is actually used. For example, if a user allows an application to access her location information, she has no way of knowing if the application will send her location to a location-based service, to advertisers, to the application developer, or to any other entity. As a result, users must blindly trust that applications will properly handle their private data. This lack of transparency forces users to blindly trust that applications will properly handle private data.

This paper describes *TaintDroid*, an extension to the Android mobile-phone platform that tracks the flow of privacy sensitive data through third-party applications. TaintDroid assumes that downloaded, third-party applications are not trusted, and monitors–in realtime–how these applications access and manipulate users' personal data. Our primary goals are to detect when sensitive data leaves the system via untrusted applications and to facilitate analysis of applications by phone users or external security services [33, 55].

Analysis of applications' behavior requires sufficient contextual information about what data leaves a device and where it is sent. Thus, TaintDroid automatically labels (taints) data from privacy-sensitive sources and transitively applies labels as sensitive data propagates through program variables, files, and interprocess messages. When tainted data are transmitted over the network, or otherwise leave the system, TaintDroid logs the data's labels, the application responsible for transmitting the data, and the data's destination. Such realtime feedback gives users and security services greater insight into what mobile applications are doing, and can potentially identify misbehaving applications.

To be practical, the performance overhead of the TaintDroid runtime must be minimal. Unlike existing solutions that rely on heavy-weight whole-system emulation [7, 57], we leveraged Android's virtualized architecture to integrate four granularities of taint propagation: variable-level, method-level, message-level, and file-level. Though the individual techniques are not new, our contributions lie in the integration of these techniques and in identifying an appropriate trade-off between performance and accuracy for resource constrained smartphones. Experiments with our prototype for Android show that tracking incurs a runtime overhead of less than 14% for a CPU-bound microbenchmark. More importantly, interactive third-party applications can be monitored with negligible perceived latency.

We evaluated the accuracy of TaintDroid using 30 randomly selected, popular Android applications that use location, camera, or microphone data. TaintDroid correctly flagged 105 instances in which these applications transmitted tainted data; of the 105, we determined that 37 were clearly legitimate. TaintDroid also revealed that 15 of the 30 applications reported users' locations to remote advertising servers. Seven applications collected the device ID and, in some cases, the phone number and the SIM card serial number. In all, two-thirds of the applications in our study used sensitive data suspiciously. Our findings demonstrate that TaintDroid can help expose potential misbehavior by third-party applications.

Like similar information-flow tracking systems [7, 57], a fundamental limitation of TaintDroid is that it can be circumvented through leaks via implicit flows. The use of implicit flows to avoid taint detection is, in and of itself, an indicator of malicious intent, and may well be detectable through other techniques such as automated static code analysis [14, 46] as we discuss in Section 8.

The rest of this paper is organized as follows: Section 2 provides a high-level overview of TaintDroid, Section 3 describes background information on the Android platform, Section 4 describes our TaintDroid design, Section 5 describes the taint sources tracked by TaintDroid, Section 6 presents results from our Android application study, Section 7 characterizes the performance of our prototype implementation, Section 8 discusses the limitations of our approach, Section 9 describes related work, and Section 10 summarizes our conclusions.

2 Approach Overview

We seek to design a framework that allows users to monitor how third-party smartphone applications handle their private data in realtime. Many smartphone applications are closed-source, therefore, static source code analysis is infeasible. Even if source code is available, runtime events and configuration often dictate information use; realtime monitoring accounts for these environment specific dependencies.

Monitoring network disclosure of privacy sensitive information on smartphones presents several challenges:

- *Smartphones are resource constrained.* The resource limitations of smartphones precludes the use of heavyweight information tracking systems such as Panorama [57].
- *Third-party applications are entrusted with several types of privacy sensitive information.* The monitoring system must distinguish multiple information types, which requires additional computation and storage.
- *Context-based privacy sensitive information is dynamic and can be difficult to identify even when sent in the clear.* For example, geographic locations are pairs of floating point numbers that frequently change and are hard to predict.
- *Applications can share information.* Limiting the monitoring system to a single application does not account for flows via files and IPC between applications, including core system applications designed to disseminate privacy sensitive information.

We use dynamic taint analysis [57, 44, 8, 61, 39] (also called "taint tracking") to monitor privacy sensitive information on smartphones. Sensitive information is first identified at a *taint source*, where a *taint marking* indicating the information type is assigned. Dynamic taint analysis tracks how labeled data impacts other data in a way that might leak the original sensitive information. This tracking is often performed at the instruction level. Finally, the impacted data is identified before it leaves the system at a *taint sink* (usually the network interface).

Existing taint tracking approaches have several limitations. First and foremost, approaches that rely on instruction-level dynamic taint analysis using whole system emulation [57, 7, 26] incur high performance penalties. Instruction-level instrumentation incurs 2-20 times slowdown [57, 7] in addition to the slowdown introduced by emulation, which is not suitable for realtime analysis. Second, developing accurate taint propagation logic has proven challenging for the x86 instruction set [40, 48]. Implementations of instruction-level tracking can experience taint explosion if the stack pointer becomes falsely tainted [49] and taint loss if complicated instructions such as CMPXCHG, REP MOV are not instrumented properly [61]. While most smartphones use the ARM instruction set, similar false positives and false negatives could arise.

Figure 1 presents our approach to taint tracking on smartphones. We leverage architectural features of virtual machine-based smartphones (e.g., Android, BlackBerry, and J2ME-based phones) to enable efficient,

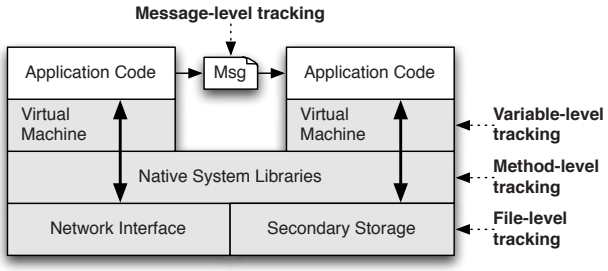


Figure 1: Multi-level approach for performance efficient taint tracking within a common smartphone architecture.

system-wide taint tracking using fine-grained labels with clear semantics. First, we instrument the VM interpreter to provide *variable-level tracking* within untrusted application code.[1] Using variable semantics provided by the interpreter provides valuable context for avoiding the taint explosion observed in the x86 instruction set. Additionally, by tracking variables, we maintain taint markings only for data and not code. Second, we use *message-level tracking* between applications. Tracking taint on messages instead of data within messages minimizes IPC overhead while extending the analysis system-wide. Third, for system-provided native libraries, we use *method-level tracking*. Here, we run native code without instrumentation and patch the taint propagation on return. These methods accompany the system and have known information flow semantics. Finally, we use *file-level* tracking to ensure persistent information conservatively retains its taint markings.

To assign labels, we take advantage of the well-defined interfaces through which applications access sensitive data. For example, all information retrieved from GPS hardware is location-sensitive, and all information retrieved from an address book database is contact-sensitive. This avoids relying on heuristics [10] or manual specification [61] for labels. We expand on information sources in Section 5.

In order to achieve this tracking at multiple granularities, our approach relies on the firmware's integrity. The taint tracking system's trusted computing base includes the virtual machine executing in userspace and any native system libraries loaded by the untrusted interpreted application. However, this code is part of the firmware, and is therefore trusted. Applications can only escape the virtual machine by executing native methods. In our target platform (Android), we modified the native library loader to ensure that applications can only load native libraries from the firmware and not those downloaded by the application. Note that an early 2010 survey of the top 50 most popular free applications in each category of the Android Market [2] (1100 applications in total) revealed that less than 4% included a `.so` file. A similar survey conducted in mid 2010 revealed this fraction increased to 5%, which indicates there is growth in the number of applications using native third-party libraries, but that the number of affected applications remains small.

In summary, we provide a novel, efficient, system-wide, multiple-marking, taint tracking design by combining multiple granularities of information tracking. While some techniques such as variable tracking within an interpreter have been previously proposed (see Section 9), to our knowledge, our approach is the first to extend such tracking system-wide. By choosing a multiple granularity approach, we balance performance and accuracy. As we show in Sections 6 and 7, our system-wide approach is both highly efficient ($\sim$14% CPU overhead and $\sim$4.4% memory overhead for simultaneously tracking 32 taint markings per data unit) and accurately detects many suspicious network packets.

3 Background: Android

Android [1] is a Linux-based, open source, mobile phone platform. Most core phone functionality is implemented as applications running on top of a customized middleware. The middleware itself is written in Java and C/C++. Applications are written in Java and compiled to a custom byte-code known as the Dalvik EXecutable (DEX) byte-code format. Each application executes within its Dalvik VM interpreter instance. Each instance executes as unique UNIX user identities to isolate applications within the Linux platform subsystem. Applications communicate via the binder IPC mechanism. Binder provides transparent message passing based on parcels. We now discuss topics necessary to understand our tracking system.

Dalvik VM Interpreter: DEX is a register-based machine language, as opposed to Java byte-code, which is stack-based. Each DEX method has its own predefined number of virtual registers (which we frequently refer to as simply "registers"). The Dalvik VM interpreter manages method registers with an internal execution state stack; the current method's registers are always on the top stack frame. These registers loosely correspond to local variables in the Java method and store primitive types and object references. All computation occurs on registers, therefore values must be loaded from and stored to class fields before use and after use. Note that DEX uses class fields for all long term storage, unlike hardware register-based machine languages (e.g., x86), which store values in arbitrary memory locations.

Native Methods: The Android middleware provides access to native libraries for performance optimization and third-party libraries such as OpenGL and Webkit. Android also uses Apache Harmony Java [3], which frequently uses system libraries (e.g., math routines). Native methods are written in C/C++ and expose functionality provided by the underlying Linux kernel and ser-

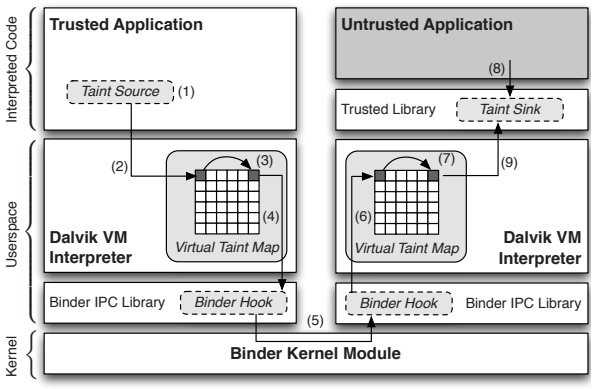


Figure 2: TaintDroid architecture within Android.

vices. They can also access Java internals, and hence are included in our trusted computing base (see Section 2).

Android contains two types of native methods: internal VM methods and JNI methods. The internal VM methods access interpreter-specific structures and APIs. JNI methods conform to Java native interface standards specifications [32], which requires Dalvik to separate Java arguments into variables using a JNI call bridge. Conversely, internal VM methods must manually parse arguments from the interpreter's byte array of arguments.

Binder IPC: All Android IPC occurs through binder. Binder is a component-based processing and IPC framework designed for BeOS, extended by Palm Inc., and customized for Android by Google. Fundamental to binder are *parcels*, which serialize both active and standard data objects. The former includes references to binder objects, which allows the framework to manage shared data objects between processes. A binder kernel module passes parcel messages between processes.

4 TaintDroid

TaintDroid is a realization of our multiple granularity taint tracking approach within Android. TaintDroid uses variable-level tracking within the VM interpreter. Multiple taint markings are stored as one *taint tag*. When applications execute native methods, variable taint tags are patched on return. Finally, taint tags are assigned to parcels and propagated through binder. Note that the Technical Report [17] version of this paper contains more implementation details.

Figure 2 depicts TaintDroid's architecture. Information is tainted (1) in a trusted application with sufficient context (e.g., the location provider). The taint interface invokes a native method (2) that interfaces with the Dalvik VM interpreter, storing specified taint markings in the virtual taint map. The Dalvik VM propagates taint tags (3) according to data flow rules as the trusted application uses the tainted information. Every interpreter instance simultaneously propagates taint tags. When the trusted application uses the tainted information in an IPC transaction, the modified binder library (4) ensures the parcel has a taint tag reflecting the combined taint markings of all contained data. The parcel is passed transparently through the kernel (5) and received by the remote untrusted application. Note that only the interpreted code is untrusted. The modified binder library retrieves the taint tag from the parcel and assigns it to all values read from it (6). The remote Dalvik VM instance propagates taint tags (7) identically for the untrusted application. When the untrusted application invokes a library specified as a taint sink (8), e.g., network send, the library retrieves the taint tag for the data in question (9) and reports the event.

Implementing this architecture requires addressing several system challenges, including: a) taint tag storage, b) interpreted code taint propagation, c) native code taint propagation, d) IPC taint propagation, and e) secondary storage taint propagation. The remainder of this section describes our design.

4.1 Taint Tag Storage

The choice of how to store taint tags influences performance and memory overhead. Dynamic taint tracking systems commonly store tags for every data byte or word [57, 7]. Tracked memory is unstructured and without content semantics. Frequently taint tags are stored in non-adjacent shadow memory [57] and tag maps [61]. TaintDroid uses variable semantics within the Dalvik interpreter. We store taint tags adjacent to variables in memory, providing spatial locality.

Dalvik has five variable types that require taint storage: method local variables, method arguments, class static fields, class instance fields, and arrays. In all cases, we store a 32-bit bitvector with each variable to encode the taint tag, allowing 32 different taint markings.

Dalvik stores method local variables and arguments on an internal stack. When an application invokes a method, a new stack frame is allocated for all local variables. Method arguments are also passed via the internal stack. Before calling a method, the callee places the arguments on the top of the stack such that they become high numbered registers in the callee's stack frame. We allocate taint tag storage by doubling the size of the stack frame allocation. Taint tags are interleaved between values such that register v_i originally accessed via $fp[i]$ is accessed as $fp[2 \cdot i]$ after modification. Note that Dalvik stores 64-bit variables as two adjacent 32-bit registers on the internal stack. While the byte-code interprets these adjacent registers as a single 64-bit value, the interpreter manages these registers as separate values. Therefore, our modified stack transparently stores and retrieves 64-bit values to and from separate 32-bit registers (at $fp[2 \cdot i]$ and $fp[2 \cdot i + 2]$). Finally, native method targets require

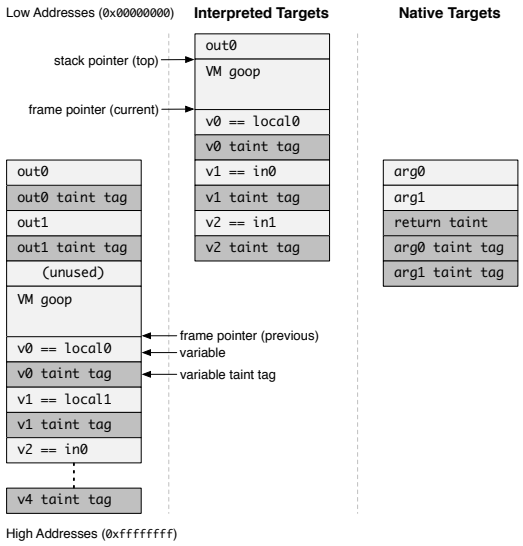


Figure 3: Modified Stack Format. Taint tags are interleaved between registers for interpreted method targets and appended for native methods. Dark grayed boxes represent taint tags.

a slightly different stack frame organization for reasons discussed in Section 4.3. The modified stack format is shown in Figure 3.

Taint tags are stored adjacent to class fields and arrays inside the VM interpreter's internal data structures. TaintDroid stores only one taint tag per array to minimize storage overhead. Per-value taint tag storage is severely inefficient for Java *String* objects, as all characters have the same tag. Unfortunately, storing one taint tag per array may result in false positives during taint propagation. For example, if untainted variable u is stored into array A at index 0 ($A[0]$) and tainted variable t is stored into $A[1]$, then array A is tainted. Later, if variable v is assigned to $A[0]$, v will be tainted, even though u was untainted. Fortunately, Java frequently uses objects, and object references are infrequently tainted (see Section 4.2), therefore this coding practice leads to less false positives.

4.2 Interpreted Code Taint Propagation

Taint tracking granularity and flow semantics influence performance and accuracy. TaintDroid implements variable-level taint tracking within the Dalvik VM interpreter. Variables provide valuable semantics for taint propagation, distinguishing data pointers from scalar values. TaintDroid primarily tracks primitive type variables (e.g., *int*, *float*, etc); however, there are cases when object references must become tainted to ensure taint propagation operates correctly; this section addresses why these cases exist. However, first we present taint tracking in the Dalvik machine language as a formal logic.

4.2.1 Taint Propagation Logic

The Dalvik VM operates on the unique DEX machine language instruction set, therefore we must design an appropriate propagation logic. We use a data flow logic, as tracking implicit flows requires static analysis and causes significant performance overhead and overestimation in tracking [29] (see Section 8). We begin by defining taint markings, taint tags, variables, and taint propagation. We then present our logic rules for DEX.

Let $\mathcal{L}$ be the universe of taint markings for a particular system. A taint tag t is a set of taint markings, $t \subseteq \mathcal{L}$. Each variable has an associated taint tag. A variable is an instance of one of the five types described in Section 4.1. We use a different representation for each type. The local and argument variables correspond to virtual registers, denoted v_x. Class field variables are denoted as f_x to indicate a field variable with class index x. Instance fields require an instance object and are denoted $v_y(f_x)$, where v_y is the instance object reference (note that both the object reference and the dereferenced value are variables). Static fields are denoted as f_x alone, which is shorthand for $S(f_x)$, where $S()$ is the static scope. Finally, $v_x[\cdot]$ denotes an array, where v_x is an array object reference variable.

Our virtual taint map function is $\tau(\cdot)$. $\tau(v)$ returns the taint tag t for variable v. $\tau(v)$ is also used to assign a taint tag to a variable. Retrieval and assignment are distinguished by the position of $\tau(\cdot)$ w.r.t. the $\leftarrow$ symbol. When $\tau(v)$ appears on the right hand side of $\leftarrow$, $\tau(v)$ retrieves the taint tag for v. When $\tau(v)$ appears on the left hand side, $\tau(v)$ assigns the taint tag for v. For example, $\tau(v_1) \leftarrow \tau(v_2)$ copies the taint tag from v_2 to v_1.

Table 1 captures our propagation logic. The table enumerates abstracted versions of the byte-code instructions specified in the DEX documentation. Register variables and class fields are referenced by v_X and f_X, respectively. R and E are the return and exception variables maintained within the interpreter, respectively. A, B, and C are constants in the byte-code. The table does not list instructions that clear the taint tag of the destination register. For example, we do not consider the *array-length* instruction to return a tainted value even if the array is tainted. Note that the array length is sometimes used to aid direct control flow propagation (e.g., Vogt et al. [53]).

4.2.2 Tainting Object References

The propagation rules in Table 1 are straightforward with two exceptions. First, taint propagation logics commonly include the taint tag of an array index during lookup to handle translation tables (e.g., ASCII/UNICODE or character case conversion). For example, consider a translation table from lowercase to upper case characters: if a tainted value "a" is used as an array index, the resulting "A" value should be tainted even though the

Table 1: DEX Taint Propagation Logic. Register variables and class fields are referenced by v_X and f_X, respectively. R and E are the return and exception variables maintained within the interpreter. A, B, and C are byte-code constants.

Op Format	Op Semantics	Taint Propagation	Description
const-op v_A C	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
move-op v_A v_B	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
move-op-R v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
return-op v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint ($\emptyset$ if void)
move-op-E v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
throw-op v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
unary-op v_A v_B	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
binary-op v_A v_B v_C	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to v_B taint $\cup$ v_C taint
binary-op v_A v_B	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
binary-op v_A v_B C	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
aput-op v_A v_B v_C	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B taint with v_A taint
aget-op v_A v_B v_C	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A taint to array and index taint
sput-op v_A f_B	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
sget-op v_A f_B	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
iput-op v_A v_B f_C	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
iget-op v_A v_B f_C	$v_A \leftarrow v_B(f_C)$	$\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B)$	Set v_A taint to field f_C and object reference taint

```
public static Integer valueOf(int i) {
  if (i < -128 || i > 127) {
    return new Integer(i); }
  return valueOfCache.CACHE [i+128];
}
static class valueOfCache {
  static final Integer[] CACHE = new Integer[256];
  static {
    for(int i=-128; i<=127; i++) {
      CACHE[i+128] = new Integer(i); } }
}
```

Figure 4: Excerpt from Android's Integer class illustrating the need for object reference taint propagation.

"A" value in the array is not. Hence, the taint logic for *aget-op* uses both the array and array index taint. Second, when the array contains object references (e.g., an *Integer* array), the index taint tag is propagated to the object reference and not the object value. Therefore, we include the object reference taint tag in the instance *get* (*iget-op*) rule.

The code listed in Figure 4 demonstrates a real instance of where object reference tainting is needed. Here, *valueOf()* returns an Integer object for a passed *int*. If the *int* argument is between -128 and 127, *valueOf()* returns reference to a statically defined Integer object. *valueOf()* is implicitly called for conversion to an object. Consider the following definition and use of a method *intProxy()*.

```
Object intProxy(int val) { return val; }
int out = (Integer) intProxy(tVal);
```

Consider the case where *tVal* is an *int* with value 1 and taint tag *TAG*. When *intProxy()* is passed *tVal*, *TAG* is propagated to *val*. When *intProxy()* returns *val*, it calls *Integer.valueOf()* to obtain an *Integer* instance corresponding to the scalar variable *val*. In this case, *Integer.valueOf()* returns a reference to the static Integer object with value 1. The *value* field (of the *Integer* class) in the object has taint tag of $\emptyset$; however, since the *aget-op* propagation rule includes the taint of the index register, the object reference has a taint tag of *TAG*. Therefore, only by including the object reference taint tag when the *value* field is read from the Integer (i.e., the *iget-op* propagation rule), will the correct taint tag of *TAG* be assigned to *out*.

4.3 Native Code Taint Propagation

Native code is unmonitored in TaintDroid. Ideally, we achieve the same propagation semantics as the interpreted counterpart. Hence, we define two *necessary postconditions* for accurate taint tracking in the Java-like environment: 1) all accessed external variables (i.e., class fields referenced by other methods) are assigned taint tags according to data flow rules; and 2) the return value is assigned a taint tag according to data flow rules. TaintDroid achieves these postconditions through an assortment of manual instrumentation, heuristics, and method profiles, depending on situational requirements.

Internal VM Methods: Internal VM methods are called directly by interpreted code, passing a pointer to an array of 32-bit register arguments and a pointer to a return value. The stack augmentation shown in Figure 3 provides access to taint tags for both Java arguments and the return value. As there are a relatively small number of internal VM methods which are infrequently added between versions,[2] we manually inspected and patched them for taint propagation as needed. We identified 185 internal VM methods in Android version 2.1; however, only 5 required patching: the *System.arraycopy()* native method for copying array contents, and several native methods implementing Java reflection.

JNI Methods: JNI methods are invoked through the JNI call bridge. The call bridge parses Java arguments and assigns a return value using the method's descriptor

string. We patched the call bridge to provide taint propagation for all JNI methods. When a JNI method returns, TaintDroid consults a method profile table for tag propagation updates. A method profile is a list of $(from, to)$ pairs indicating flows between variables, which may be method parameters, class variables, or return values. Enumerating the information flows for all JNI methods is a time consuming task best completed automatically using source code analysis (a task we leave for future work). We currently include an additional propagation heuristic patch. The heuristic is conservative for JNI methods that only operate on primitive and String arguments and return values. It assigns the union of the method argument taint tags to the taint tag of the return value. While the heuristic has false negatives for methods using objects, it covers many existing methods.

We performed a survey of the JNI methods included in the official Android source code (version 2.1) to determine specific properties. We found 2,844 JNI methods with a Java interface and C or C++ implementation.[3] Of these methods, 913 did not reference objects (as arguments, return value, or method body) and hence are automatically covered by our heuristic. The remaining methods may or may not have information flows that produce false negatives. Currently, we define method profiles as needed. For example, methods in the IBM *NativeConverter* class require propagation for conversion between character and byte arrays.

4.4 IPC Taint Propagation

Taint tags must propagate between applications when they exchange data. The tracking granularity affects performance and memory overhead. TaintDroid uses message-level taint tracking. A message taint tag represents the upper bound of taint markings assigned to variables contained in the message. We use message-level granularity to minimize performance and storage overhead during IPC.

We chose to implement message-level over variable-level taint propagation, because in a variable-level system, a devious receiver could game the monitoring by unpacking variables in a different way to acquire values without taint propagation. For example, if an IPC parcel message contains a sequence of scalar values, the receiver may unpack a string instead, thereby acquiring values without propagating all the taint tags on scalar values in the sequence. Hence, to prevent applications from removing taint tags in this way, the current implementation protects taint tags at the message-level.

Message-level taint propagation for IPC leads to false positives. Similar to arrays, all data items in a parcel share the same taint tag. For example, Section 8 discusses limitations for tracking the IMSI that results from passing as portions the value as configuration parameters in parcels. Future implementations will consider word-level taint tags along with additional consistency checks to ensure accurate propagation for unpacked variables. However, this additional complexity will negatively impact IPC performance.

4.5 Secondary Storage Taint Propagation

Taint tags may be lost when data is written to a file. Our design stores one taint tag per file. The taint tag is updated on file write and propagated to data on file read. TaintDroid stores file taint tags in the file system's extended attributes. To do this, we implemented extended attribute support for Android's host file system (YAFFS2) and formatted the removable SDcard with the ext2 file system. As with arrays and IPC, storing one taint tag per file leads to false positives and limits the granularity of taint markings for information databases (see Section 5). Alternatively, we could track taint tags at a finer granularity at the expense of added memory and performance overhead.

4.6 Taint Interface Library

Taint sources and sinks defined within the virtualized environment must communicate taint tags with the tracking system. We abstract the taint source and sink logic into a single taint interface library. The interface performs two functions: 1) add taint markings to variables; and 2) retrieve taint markings from variables. The library only provides the ability to add and not set or clear taint tags, as such functionality could be used by untrusted Java code to remove taint markings.

Adding taint tags to arrays and strings via internal VM methods is straightforward, as both are stored in data objects. Primitive type variables, on the other hand, are stored on the interpreter's internal stack and disappear after a method is called. Therefore, the taint library uses the method return value as a means of tainting primitive type variables. The developer passes a value or variable into the appropriate add taint method (e.g., *addTaintInt()*) and the returned variable has the same value but additionally has the specified taint tag. Note that the stack storage does not pose complications for taint tag retrieval.

5 Privacy Hook Placement

Using TaintDroid for privacy analysis requires identifying privacy sensitive sources and instrumenting taint sources within the operating system. Historically, dynamic taint analysis systems assume taint source and sink placement is trivial. However, complex operating systems such as Android provide applications information in a variety of ways, e.g., direct access, and service interface. Each potential type of privacy sensitive information must be studied carefully to determine the best method of defining the taint source.

Taint sources can only add taint tags to memory for which TaintDroid provides tag storage. Currently, taint source and sink placement is limited to variables in interpreted code, IPC messages, and files. This section discusses how valuable taint sources and sinks can be implemented within these restrictions. We generalize such taint sources based on information characteristics.

Low-bandwidth Sensors: A variety of privacy sensitive information types are acquired through low-bandwidth sensors, e.g., location and accelerometer. Such information often changes frequently and is simultaneously used by multiple applications. Therefore, it is common for a smartphone OS to multiplex access to low-bandwidth sensors using a manager. This sensor manager represents an ideal point for taint source hook placement. For our analysis, we placed hooks in Android's LocationManager and SensorManager applications.

High-bandwidth Sensors: Privacy sensitive information sources such as the microphone and camera are high-bandwidth. Each request from the sensor frequently returns a large amount of data that is only used by one application. Therefore, the smartphone OS may share sensor information via large data buffers, files, or both. When sensor information is shared via files, the file must be tainted with the appropriate tag. Due to flexible APIs, we placed hooks for both data buffer and file tainting for tracking microphone and camera information.

Information Databases: Shared information such as address books and SMS messages are often stored in file-based databases. This organization provides a useful unambiguous taint source similar to hardware sensors. By adding a taint tag to such database files, all information read from the file will be automatically tainted. We used this technique for tracking address book information. Note that while TaintDroid's file-level granularity was appropriate for these valuable information sources, others may exist for which files are too coarse grained. However, we have not yet encountered such sources.

Device Identifiers: Information that uniquely identifies the phone or the user is privacy sensitive. Not all personally identifiable information can be easily tainted. However, the phone contains several easily tainted identifiers: the phone number, SIM card identifiers (IMSI, ICC-ID), and device identifier (IMEI) are all accessed through well-defined APIs. We instrumented the APIs for the phone number, ICC-ID, and IMEI. An IMSI taint source has inherent limitations discussed in Section 8.

Network Taint Sink: Our privacy analysis identifies when tainted information transmits out the network interface. The VM interpreter-based approach requires the taint sink to be placed within interpreted code. Hence, we instrumented the Java framework libraries at the point the native socket library is invoked.

6 Application Study

This section reports on an application study that uses TaintDroid to analyze how 30 popular third-party Android applications use privacy sensitive user data. Existing applications acquire a variety of user data along with permissions to access the Internet. Our study finds that two thirds of these applications expose detailed location data, the phone's unique ID, and the phone number using the combination of the seemingly innocuous access permissions granted at install. This finding was made possible by TaintDroid's ability to monitor runtime *access* of sensitive user data and to precisely relate the monitored accesses with the *data exposure* by applications.

6.1 Experimental Setup

An early 2010 survey of the 50 most popular free applications in each category of the Android Market [2] (1,100 applications, in total) revealed that roughly a third of the applications (358 of the 1,100 applications) require Internet permissions along with permissions to access either location, camera, or audio data. From this set, we randomly selected 30 popular applications (an 8.4% sample size), which span twelve categories. Table 2 enumerates these applications along with permissions they request at install time. Note that this does not reflect actual access or use of sensitive data.

We studied each of the thirty downloaded applications by starting the application, performing any initialization or registration that was required, and then manually exercising the functionality offered by the application. We recorded system logs including detailed information from TaintDroid: tainted binder messages, tainted file output, and tainted network messages with the remote address. The overall experiment (conducted in May 2010) lasted slightly over 100 minutes, generating 22,594 packets (8.6MB) and 1,130 TCP connections. To verify our results, we also logged the network traffic using tcpdump on the WiFi interface and repeated experiments on multiple Nexus One phones, running the same version of TaintDroid built on Android 2.1. Though the phones used for experiments had a valid SIM card installed, the SIM card was inactivate, forcing all the packets to be transmitted via the WiFi interface. The packet trace was used only to verify the exposure of tainted data flagged by TaintDroid.

In addition to the network trace, we also noted whether applications acquired user consent (either explicit or implicit) for exporting sensitive information. This provides additional context information to identify possible privacy violations. For example, by selecting the "use my location" option in a weather application, the user implicitly consents to disclosing geographic coordinates to the weather server.

Table 2: Applications grouped by the requested permissions (L: location, C: camera, A: audio, P: phone state). Android Market categories are indicated in parenthesis, showing the diversity of the studied applications.

Applications	#	Permissions*			
		L	C	A	P
The Weather Channel (News & Weather); Cestos, Solitaire (Game); Movies (Entertainment); Babble (Social); Manga Browser (Comics)	6	x			
Bump, Wertago (Social); Antivirus (Communication); ABC — Animals, Traffic Jam, Hearts, Blackjack, (Games); Horoscope (Lifestyle); 3001 Wisdom Quotes Lite, Yellow Pages (Reference); Dastelefonbuch, Astrid (Productivity), BBC News Live Stream (News & Weather); Ringtones (Entertainment)	14	x			x
Layer (Productivity); Knocking (Social); Barcode Scanner, Coupons (Shopping); Trapster (Travel); Spongebob Slide (Game); ProBasketBall (Sports)	7	x	x		x
MySpace (Social); ixMAT (Shopping)	2		x		
Evernote (Productivity)	1	x	x	x	

* All listed applications also require access to the Internet.

Table 3: Potential privacy violations by 20 of the studied applications. Note that three applications had multiple violations, one of which had a violation in all three categories.

Observed Behavior (# of apps)	Details
Phone Information to Content Servers (2)	2 apps sent out the phone number, IMSI, and ICC-ID along with the geo-coordinates to the app's content server.
Device ID to Content Servers (7)*	2 Social, 1 Shopping, 1 Reference and three other apps transmitted the IMEI number to the app's content server.
Location to Advertisement Servers (15)	5 apps sent geo-coordinates to ad.qwapi.com, 5 apps to admob.com, 2 apps to ads.mobclix.com (1 sent location both to admob.com and ads.mobclix.com) and 4 apps sent location† to data.flurry.com.

* TaintDroid flagged nine applications in this category, but only seven transmitted the raw IMEI without mentioning such practice in the EULA.

†To the best of our knowledge, the binary messages contained tainted location data (see the discussion below).

6.2 Findings

Table 3 summarizes our findings. TaintDroid flagged 105 TCP connections as containing tainted privacy sensitive information. We manually labeled each message based on available context, including remote server names and temporally relevant application log messages. We used remote hostnames as an indication of whether data was being sent to a server providing application functionality or to a third party. Frequently, messages contained plaintext that aided categorization, e.g., an HTTP GET request containing geographic coordinates. However, 21 flagged messages contained binary data. Our investigation indicates these messages were generated by the Google Maps for Mobile [21] and FlurryAgent [20] APIs and contained tainted privacy sensitive data. These conclusions are supported by message transmissions immediately after the application received a tainted parcel from the system location manager. We now expand on our findings for each category and reflect on potential privacy violations.

Phone Information: Table 2 shows that 21 out of the 30 applications require permissions to read phone state and the Internet. We found that 2 of the 21 applications transmitted to their server (1) the device's phone number, (2) the IMSI which is a unique 15-digit code used to identify an individual user on a GSM network, and (3) the ICC-ID number which is a unique SIM card serial number. We verified messages were flagged correctly by inspecting the plaintext payload.[4] In neither case was the user informed that this information was transmitted off the phone.

This finding demonstrates that Android's coarse-grained access control provides insufficient protection against third-party applications seeking to collect sensitive data. Moreover, we found that one application transmits the phone information *every time* the phone boots. While this application displays a terms of use on first use, the terms of use does not specify collection of this highly sensitive data. Surprisingly, this application transmits the phone data immediately after install, before first use.

Device Unique ID: The device's IMEI was also exposed by applications. The IMEI uniquely identifies a specific mobile phone and is used to prevent a stolen handset from accessing the cellular network. TaintDroid flags indicated that nine applications transmitted the IMEI. Seven out of the nine applications either do not present an End User License Agreement (EULA) or do not specify IMEI collection in the EULA. One of the seven applications is a popular social networking application and another is a location-based search application. Further-

more, we found two of the seven applications include the IMEI when transmitting the device's geographic coordinates to their content server, potentially repurposing the IMEI as a client ID.

In comparison, two of the nine applications treat the IMEI with more care, thus we do not classify them as potential privacy violators. One application displays a privacy statement that clearly indicates that the application collects the device ID. The other uses the hash of the IMEI instead of the number itself. We verified this practice by comparing results from two different phones.

Location Data to Advertisement Servers: Half of the studied applications exposed location data to third-party advertisement servers without requiring implicit or explicit user consent. Of the fifteen applications, only two presented a EULA on first run; however neither EULA indicated this practice. Exposure of location information occurred both in plaintext and in binary format. The latter highlights TaintDroid's advantages over simple pattern-based packet scanning. Applications sent location data in plaintext to admob.com, ad.qwapi.com, ads.mobclix.com (11 applications) and in binary format to FlurryAgent (4 applications). The plaintext location exposure to AdMob occurred in the HTTP GET string:

```
...&s=a14a4a93f1e4c68&..&t=062A1CB1D476DE85
B717D9195A6722A9&d%5Bcoord%5D=47.6612278900
00006%2C-122.31589477&...
```

Investigating the AdMob SDK revealed the `s=` parameter is an identifier unique to an application publisher, and the `coord=` parameter provides the geographic coordinates.

For FlurryAgent, we confirmed location exposure by the following sequence of events. First, a component named "FlurryAgent" registers with the location manager to receive location updates. Then, TaintDroid log messages show the application receiving a tainted parcel from the location manager. Finally, the application reports "sending report to `http://data.flurry.com/aar.do`" after receiving the tainted parcel.

Our experimentation indicates these fifteen applications collect location data and send it to advertisement servers. In some cases, location data was transmitted to advertisement servers even when no advertisement was displayed in the application. However, we note that TaintDroid helped us verify that three of the studied applications (not included in the Table 3) only transmitted location data per user's request to pull localized content from their servers. This finding demonstrates the importance of monitoring exercised functionality of an application that reflects how the application *actually* uses or abuses the granted permissions.

Legitimate Flags: Out of 105 connections flagged by TaintDroid, 37 were deemed clearly legitimate use. The flags resulted from four applications and the OS itself while using the Google Maps for Mobile (GMM) API. The TaintDroid logs indicate an HTTP request with the "User-Agent: GMM ..." header, but a binary payload. Given that GMM functionality includes downloading maps based on geographic coordinates, it is obvious that TaintDroid correctly identified location information in the payload. Our manual inspection of each message along with the network packet trace confirmed that there were no false positives. We note that there is a possibility of false negatives, which is difficult to verify with the lack of the source code of the third-party applications.

Summary: Our study of 30 popular applications shows the effectiveness of the TaintDroid system in accurately tracking applications' use of privacy sensitive data. While monitoring these applications, TaintDroid generated no false positives (with the exception of the IMSI taint source which we disabled for experiments, see Section 8). The flags raised by TaintDroid helped to identify potential privacy violations by the tested applications. Half of the studied applications share location data with advertisement servers. Approximately one third of the applications expose the device ID, sometimes with the phone number and the SIM card serial number. The analysis was simplified by the taint tag provided by TaintDroid that precisely describes which privacy relevant data is included in the payload, especially for binary payloads. We also note that there was almost no perceived latency while running experiments with TaintDroid.

7 Performance Evaluation

We now study TaintDroid's taint tracking overhead. Experiments were performed on a Google Nexus One running Android OS version 2.1 modified for TaintDroid. Within the interpreted environment, TaintDroid incurs the same performance and memory overhead regardless of the existence of taint markings. Hence, we only need to ensure file access includes appropriate taint tags.

7.1 Macrobenchmarks

During the application study, we anecdotally observed limited performance overhead. We hypothesize that this is because: 1) most applications are primarily in a "wait state," and 2) heavyweight operations (e.g., screen updates and webpage rendering) occur in unmonitored native libraries.

To gain further insight into perceived overhead, we devised five macrobenchmarks for common high-level smartphone operations. Each experiment was measured 50 times and observed 95% confidence intervals at least an order of magnitude less than the mean. In each case, we excluded the first run to remove unrelated initialization costs. Experimental results are shown in Table 4.

Application Load Time: The application load time measures from when Android's Activity Manager re-

Table 4: Macrobenchmark Results

	Android	TaintDroid
App Load Time	63 ms	65 ms
Address Book (create)	348 ms	367 ms
Address Book (read)	101 ms	119 ms
Phone Call	96 ms	106 ms
Take Picture	1718 ms	2216 ms

ceives a command to start an activity component to the time the activity thread is displayed. This time includes application resolution by the Activity Manager, IPC, and graphical display. TaintDroid adds only 3% overhead, as the operation is dominated by native graphics libraries.

Address Book: We built a custom application to create, read, and delete entries for the phone's address book, exercising both file read and write. Create used three SQL transactions while read used two SQL transactions. The subsequent delete operation was lazy, returning in 0 ms, and hence was excluded from our results. TaintDroid adds approximately 5.5% and 18% overhead for address book entry creates and reads, respectively. The additional overhead for reads can be attributed to file taint propagation. The data is not tainted before create, hence no file propagation is needed. Note that the user experiences less than 20 ms overhead when creating or viewing a contact.

Phone Call: The phone call benchmark measured the time from pressing "dial" to the point at which the audio hardware was reconfigured to "in call" mode. TaintDroid only adds 10 ms per phone call setup ($\sim$10% overhead), which is significantly less than call setup in the network, which takes on the order of seconds.

Take Picture: The picture benchmark measures from the time the user presses the "take picture" button until the preview display is re-enabled. This measurement includes the time to capture a picture from the camera and save the file to the SDcard. TaintDroid adds 498 ms to the 1718 ms needed by Android to take a picture (an overhead of 29%). A portion of this overhead can be attributed to to additional file operations required for taint propagation (one *getxattr*/*setxattr* pair per written data buffer). Note that some of this overhead can be reduced by eliminating redundant taint propagation. That is, only the taint tag for the first data buffer written to file needs to be propagated. For example, the current taint tag could be associated with the file descriptor.

7.2 Java Microbenchmark

Figure 5 shows the execution time results of a Java microbenchmark. We used an Android port of the standard CaffeineMark 3.0 [43]. CaffeineMark uses an internal scoring metric only useful for relative comparisons.

The results are consistent with implementation-

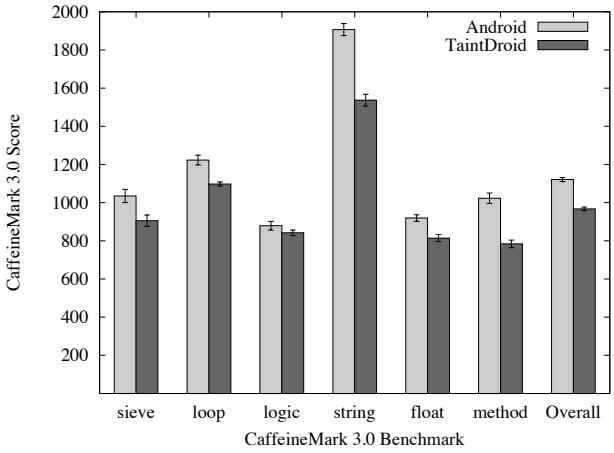


Figure 5: Microbenchmark of Java overhead. Error bars indicate 95% confidence intervals.

specific expectations. The overhead incurred by TaintDroid is smallest for the benchmarks dominated by arithmetic and logic operations. The taint propagation for these operations is simple, consisting of an additional copy of spatially local memory. The string benchmark, on the other hand, experiences the greatest overhead. This is most likely due to the additional memory comparisons that occur when the JNI propagation heuristic checks for string objects in method prototypes.

The "overall" results indicate cumulative score across individual benchmarks. CaffeineMark documentation states that scores roughly correspond to the number of Java instructions executed per second. Here, the unmodified Android system had an average score of 1121, and TaintDroid measured 967. TaintDroid has a 14% overhead with respect to the unmodified system.

We also measured memory consumption during the CaffeineMark benchmark. The benchmark consumed 21.28 MB on the unmodified system and 22.21 MB while running on TaintDroid, indicating a 4.4% memory overhead. Note that much of an Android process's memory is used by the zygote runtime environment. These native library memory pages are shared between applications to reduce the overall system memory footprint and require taint tracking. Given that TaintDroid stores 32 taint markings (4 bytes) for each 32-bit variable in the interpreted environment (regardless of taint state), this overhead is expected.

7.3 IPC Microbenchmark

The IPC benchmark considers overhead due to the parcel modifications. For this experiment, we developed client and service applications that perform binder transactions as fast as possible. The service manipulates account objects (a username string and a balance integer) and provides two interfaces: *setAccount()* and *getAc-*

Table 5: IPC Throughput Test (10,000 msgs).

	Android	TaintDroid
Time (s)	8.58	10.89
Memory (client)	21.06MB	21.88MB
Memory (service)	18.92MB	19.48MB

count(). The experiment measures the time for the client to invoke each interface pair 10,000 times.

Table 5 summarizes the results of the IPC benchmark. TaintDroid was 27% slower than Android. TaintDroid only adds four bytes to each IPC object, therefore overhead due to data size is unlikely. The more likely cause of the overhead is the continual copying of taint tags as values are marshalled into and out of the parcel byte buffer. Finally, TaintDroid used 3.5% more memory than Android, which is comparable to the consumption observed during the CaffeineMark benchmarks.

8 Discussion

Approach Limitations: TaintDroid only tracks data flows (i.e., explicit flows) and does not track control flows (i.e., implicit flows) to minimize performance overhead. Section 6 shows that TaintDroid can track applications' expected data exposure and also reveal suspicious actions. However, applications that are truly malicious can game our system and exfiltrate privacy sensitive information through control flows. Fully tracking control flow requires static analysis [14, 37], which is not applicable to analyzing third-party applications whose source code is unavailable. Direct control flows can be tracked dynamically if a taint scope can be determined [53]; however, DEX does not maintain branch structures that TaintDroid can leverage. On-demand static analysis to determine method control flow graphs (CFGs) provides this context [39]; however, TaintDroid does not currently perform such analysis in order to avoid false positives and significant performance overhead. Our data flow taint propagation logic is consistent with existing, well known, taint tracking systems [7, 57]. Finally, once information leaves the phone, it may return in a network reply. TaintDroid cannot track such information.

Implementation Limitations: Android uses the Apache Harmony [3] implementation of Java with a few custom modifications. This implementation includes support for the *PlatformAddress* class, which contains a native address and is used by *DirectBuffer* objects. The file and network IO APIs include write and read "direct" variants that consume the native address from a *DirectBuffer*. TaintDroid does not currently track taint tags on *DirectBuffer* objects, because the data is stored in opaque native data structures. Currently, TaintDroid logs when a read or write "direct" variant is used, which anecdotally occurred with minimal frequency. Similar implementation limitations exist with the *sun.misc.Unsafe* class, which also operates on native addresses.

Taint Source Limitations: While TaintDroid is very effective for tracking sensitive information, it causes significant false positives when the tracked information contains configuration identifiers. For example, the IMSI numeric string consists of a Mobile Country Code (MCC), Mobile Network Code (MNC), and Mobile Station Identifier Number (MSIN), which are all tainted together.[5] Android uses the MCC and MNC extensively as configuration parameters when communicating other data. This causes all information in a parcel to become tainted, eventually resulting in an explosion of tainted information. Thus, for taint sources that contain configuration parameters, tainting individual variables within parcels is more appropriate. However, as our analysis results in Section 6 show, message-level taint tracking is effective for the majority of our taint sources.

9 Related Work

Mobile phone host security is a growing concern. OS-level protections such as Kirin [18], Saint [42], and Security-by-Contract [15] provide enhanced security mechanisms for Android and Windows Mobile. These approaches prevent access to sensitive information; however, once information enters the application, no additional mediation occurs. In systems with larger displays, a graphical widget [27] can help users visualize sensor access policies. Mulliner et al. [36] provide information tracking by labeling smartphone processes based on the interfaces they access, effectively limiting access to future interfaces based on acquired labels.

Decentralized information flow control (DIFC) enhanced operating systems such as Asbestos [52] and HiStar [60] label processes and enforce access control based on Denning's lattice model for information flow security [13]. Flume [30] provides similar enhancements for legacy OS abstractions. DEFCon [34] uses a logic similar to these DIFC OSes, but focuses on events and modifies a Java runtime with lightweight isolation. Related to these system-level approaches, PRECIP [54] labels both processes and shared kernel objects such as the clipboard and display buffer. However, these process-level information flow models are coarse grained and cannot track sensitive information *within* untrusted applications.

Tools that analyze applications for privacy sensitive information leaks include Privacy Oracle [28] and TightLip [59]. These tools investigate applications while treating them as a black box, thus enabling analysis of off-the-shelf applications. However, this black-box analysis tool becomes ineffective when applications use encryption prior to releasing sensitive information.

Language-based information flow security [46] extends existing programming languages by labeling vari-

ables with security attributes. Compilers use the security labels to generate security proofs, e.g., Jif [37, 38] and SLam [24]. Laminar [45] provides DIFC guarantees based on programmer defined security regions. However, these languages require careful development and are often incompatible with legacy software designs [25].

Dynamic taint analysis provides information tracking for legacy programs. The approach has been used to enhance system integrity (e.g., defend against software attacks [41, 44, 8]) and confidentiality (e.g., discover privacy exposure [57, 16, 61]), as well as track Internet worms [9]. Dynamic tracking approaches range from whole-system analysis using hardware extensions [51, 11, 50] and emulation environments [7, 57] to per-process tracking using dynamic binary translation (DBT) [6, 44, 8, 61]. The performance and memory overhead associated with dynamic tracking has resulted in an array of optimizations, including optimizing context switches [44], on-demand tracking [26] based on hypervisor introspection, and function summaries for code with known information flow properties [61]. If source code is available, significant performance improvements can be achieved by automatically instrumenting legacy programs with dynamic tracking functionality [56, 31]. Automatic instrumentation has also been performed on x86 binaries [47], providing a compromise between source code translation and DBT. Our TaintDroid design was inspired by these prior works, but addressed different challenges unique to mobile phones. To our knowledge, TaintDroid is the first taint tracking system for a mobile phone and is the first dynamic taint analysis system to achieve practical system-wide analysis through the integration of tracking multiple data object granularities.

Finally, dynamic taint analysis has been applied to virtual machines and interpreters. Haldar et al. [22] instrument the Java String class with taint tracking to prevent SQL injection attacks. WASP [23] has similar motivations; however, it uses positive tainting of individual characters to ensure the SQL query contains only high-integrity substrings. Chandra and Franz [5] propose fine-grained information flow tracking within the JVM and instrument Java byte-code to aid control flow analysis. Similarly, Nair et al. [39] instrument the Kaffe JVM. Vogt et al. [53] instrument a Javascript interpreter to prevent cross-site scripting attacks. Xu et al. [56] automatically instrument the PHP interpreter source code with dynamic information tracking to prevent SQL injection attacks. Finally, the Resin [58] environment for PHP and Python uses data flow tracking to prevent an assortment of Web application attacks. When data leaves the interpreted environment, Resin implements filters for files and SQL databases to serialize and de-serialize objects and policy with byte-level granularity. TaintDroid's interpreted code taint propagation bears similarity to some of these works. However, TaintDroid implements system-wide information flow tracking, seamlessly connecting interpreter taint tracking with a range of operating system sharing mechanisms.

10 Conclusions

While some mobile phone operating systems allow users to control applications' access to sensitive information, such as location sensors, camera images, and contact lists, users lack visibility into how applications use their private data. To address this, we present TaintDroid, an efficient, system-wide information flow tracking tool that can simultaneously track multiple sources of sensitive data. A key design goal of TaintDroid is efficiency, and TaintDroid achieves this by integrating four granularities of taint propagation (variable-level, message-level, method-level, and file-level) to achieve a 14% performance overhead on a CPU-bound microbenchmark.

We also used our TaintDroid implementation to study the behavior of 30 popular third-party applications, chosen at random from the Android Marketplace. Our study revealed that two-thirds of the applications in our study exhibit suspicious handling of sensitive data, and that 15 of the 30 applications reported users' locations to remote advertising servers. Our findings demonstrate the effectiveness and value of enhancing smartphone platforms with monitoring tools such as TaintDroid.

Acknowledgments

We would like to thank Intel Labs, Berkeley and Seattle for its support and feedback during the design and prototype implementation of this work. We thank Jayanth Kannon, Stuart Schechter, and Ben Greenstein for their feedback during the writing of this paper. We also thank Kevin Butler, Stephen McLaughlin, Machigar Ongtang, and the SIIS lab as a whole for their helpful comments. This material is based upon work supported by the National Science Foundation. William Enck and Patrick McDaniel were partially supported by NSF Grant No. CNS-0905447, CNS-0721579 and CNS-0643907. Landon Cox and Peter Gilbert were partially supported by NSF CAREER Award CNS-0747283. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Android. `http://www.android.com`.

[2] Android Market. `http://market.android.com`.

[3] Apache Harmony – Open Source Java Platform. `http://harmony.apache.org`.

[4] APPLE, INC. Apples App Store Downloads Top Three Billion. http://www.apple.com/pr/library/2010/01/05appstore.html, January 2010.

[5] CHANDRA, D., AND FRANZ, M. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)* (December 2007).

[6] CHENG, W., ZHAO, Q., YU, B., AND HIROSHIGE, S. TaintTrace: Efficient Flow Tracing with Dyanmic Binary Rewriting. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)* (June 2006), pp. 749–754.

[7] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the 13th USENIX Security Symposium* (August 2004).

[8] CLAUSE, J., LI, W., AND ORSO, A. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 international symposium on Software testing and analysis* (2007), pp. 196–206.

[9] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the ACM Symposium on Operating Systems Principles* (2005).

[10] COX, L. P., AND GILBERT, P. RedFlag: Reducing Inadvertent Leaks by Personal Machines. Tech. Rep. TR-2009-02, Duke University, 2009.

[11] CRANDALL, J. R., AND CHONG, F. T. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the International Symposium on Microarchitecture* (December 2004), pp. 221–232.

[12] DAVIES, C. iPhone spyware debated as app library "phones home". http://www.slashgear.com/iphone-spyware-debated-as-app-library-phones-home-1752491/, August 17, 2009.

[13] DENNING, D. E. A Lattice Model of Secure Information Flow. *Communications of the ACM 19*, 5 (May 1976), 236–243.

[14] DENNING, D. E., AND DENNING, P. J. Certification of Programs for Secure Information Flow. *Communications of the ACM 20*, 7 (July 1977).

[15] DESMET, L., JOOSEN, W., MASSACCI, F., PHILIPPAERTS, P., PIESSENS, F., SIAHAAN, I., AND VANOVERBERGHE, D. Security-by-contract on the .NET platform. *Information Security Technical Report 13*, 1 (January 2008), 25–32.

[16] EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., AND SONG, D. Dyanmic Spyware Analysis. In *Proceedings of the USENIX Annual Technical Conference* (June 2007), pp. 233–246.

[17] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. Tech. Rep. NAS-TR-0120-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, August 2010.

[18] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (November 2009).

[19] FITZPATRICK, M. Mobile that allows bosses to snoop on staff developed. BBC News, March 2010. http://news.bbc.co.uk/2/hi/technology/8559683.stm.

[20] Flurry Mobile Application Analytics. http://www.flurry.com/product/technical-info.html.

[21] Google Maps for Mobile. http://www.google.com/mobile/products/maps.html.

[22] HALDAR, V., CHANDRA, D., AND FRANZ, M. Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)* (December 2005), pp. 303–311.

[23] HALFOND, W. G., ORSO, A., AND MANOLIOS, P. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering 34*, 1 (2008), 65–81.

[24] HEINTZE, N., AND RIECKE, J. G. The SLam Calculus: Programming with Secrecy and Integrity. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)* (1998), pp. 365–377.

[25] HICKS, B., AHMADIZADEH, K., AND MCDANIEL, P. Understanding practical application development in security-typed languages. In *22st Annual Computer Security Applications Conference (ACSAC)* (2006), pp. 153–164.

[26] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. Practical Taint-Based Protection using Demand Emulation. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (2006), pp. 29–41.

[27] HOWELL, J., AND SCHECHTER, S. What You See is What they Get: Protecting users from unwanted use of microphones, camera, and other sensors. In *Proceedings of Web 2.0 Security and Privacy Workshop* (2010).

[28] JUNG, J., SHETH, A., GREENSTEIN, B., WETHERALL, D., MAGANIS, G., AND KOHNO, T. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *Proceedings of ACM CCS* (2008).

[29] KING, D., HICKS, B., HICKS, M., AND JAEGER, T. Implicit Flows: Can't Live with 'Em, Can't Live without 'Em. In *Proceedings of the International Conference on Information Systems Security* (2008).

[30] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information Flow Control for Standard OS Abstractions. In *Proceedings of ACM Symposium on Operating Systems Principles* (2007).

[31] LAM, L. C., AND CKER CHIUEH, T. A General Dynamic Information Flow Tracking Framework for Security Applications. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2006).

[32] LIANG, S. *Java Native Interface: Programmer's Guide and Specification*. Prentice Hall PTR, 1999.

[33] LOOKOUT. Introducing the App Genome Project. http://blog.mylookout.com/2010/07/introducing-the-app-genome-project/, July 2010.

[34] MIGLIAVACCA, M., PAPAGIANNIS, I., EYERS, D. M., SHAND, B., BACON, J., AND PIETZUCH, P. DEFCon: High-Performance Event Processing with Information Security. In *PROCEEDINGS of the USENIX Annual Technical Conference* (2010).

[35] MOREN, D. Retrievable iPhone numbers mean potential privacy issues. http://www.macworld.com/article/143047/2009/09/phone_hole.html, September 29, 2009.

[36] MULLINER, C., VIGNA, G., DAGON, D., AND LEE, W. Using Labeling to Prevent Cross-Service Attacks Against Smart Phones. In *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2006).

[37] MYERS, A. C. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the ACM Symposium on Principles of Programming Langauges (POPL)* (January 1999).

[38] MYERS, A. C., AND LISKOV, B. Protecting Privacy Using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology 9*, 4 (October 2000), 410–442.

[39] NAIR, S. K., SIMPSON, P. N., CRISPO, B., AND TANENBAUM, A. S. A Virtual Machine Based Information Flow Control System for Policy Enforcement. In *the 1st International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM)* (2007).

[40] NEWSOME, J., MCCAMANT, S., AND SONG, D. Measuring channel capacity to distinguish undue influence. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (2009).

[41] NEWSOME, J., AND SONG, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. of Network and Distributed System Security Symposium* (2005).

[42] ONGTANG, M., MCLAUGHLIN, S., ENCK, W., AND MCDANIEL, P. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)* (2009).

[43] PENDRAGON SOFTWARE CORPORATION. CaffeineMark 3.0. `http://www.benchmarkhq.ru/cm30/`.

[44] QIN, F., WANG, C., LI, Z., SEOP KIM, H., ZHOU, Y., AND WU, Y. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (2006), pp. 135–148.

[45] ROY, I., PORTER, D. E., BOND, M. D., MCKINLEY, K. S., AND WITCHEL, E. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In *Proceedings of Programming Language Design and Implementation* (2009).

[46] SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication 21*, 1 (January 2003), 5–19.

[47] SAXENA, P., SEKAR, R., AND PURANIK, V. Efficient Fine-Grained Binary Instrumentation with Applications to Taint-Tracking. In *Proceedings of the IEEE/ACM symposium on Code Generation and Optimization (CGO)* (2008).

[48] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy* (2010).

[49] SLOWINSKA, A., AND BOS, H. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (April 2009), pp. 61–74.

[50] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of Architectural Support for Programming Languages and Operating Systems* (2004).

[51] VACHHARAJANI, N., BRIDGES, M. J., CHANG, J., RANGAN, R., OTTONI, G., BLOME, J. A., REIS, G. A., VACHHARAJANI, M., AND AUGUST, D. I. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture* (2004), pp. 243–254.

[52] VANDEBOGART, S., EFSTATHOPOULOS, P., KOHLER, E., KROHN, M., FREY, C., ZIEGLER, D., KAASHOEK, F., MORRIS, R., AND MAZIÈRES, D. Labels and Event Processes in the Asbestos Operating System. *ACM Transactions on Computer Systems (TOCS) 25*, 4 (December 2007).

[53] VOGT, P., NENTWICH, F., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proc. of Network & Distributed System Security* (2007).

[54] WANG, X., LI, Z., LI, N., AND CHOI, J. Y. PRECIP: Towards Practical and Retrofittable Confidential Information Protection. In *Proceedings of 15th Network and Distributed System Security Symposium (NDSS)* (2008).

[55] WhatApp. `http://www.whatapp.org`. Accessed April 2010.

[56] XU, W., BHATKAR, S., AND SEKAR, R. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the USENIX Security Symposium* (August 2006), pp. 121–136.

[57] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of ACM Computer and Communications Security* (2007).

[58] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving Application Security with Data Flow Assertions. In *Proceedings of the ACM Symposium on Operating Systems Principles* (Oct. 2009).

[59] YUMEREFENDI, A. R., MICKLE, B., AND COX, L. P. TightLip: Keeping Applications from Spilling the Beans. In *Proceedings of the 4th USENIX Symposium on Network Systems Design & Implementation (NSDI)* (2007).

[60] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI)* (2006).

[61] ZHU, D., JUNG, J., SONG, D., KOHNO, T., AND WETHERALL, D. Privacy Scope: A Precise Information Flow Tracking System For Finding Application Leaks. Tech. Rep. EECS-2009-145, Department of Computer Science, UC Berkeley, 2009.

Notes

[1] A similar approach can be applied to just-in-time compilation by inserting tracking code within the generated binary.

[2] Only 11 internal VM methods were added between versions 1.5 and 2.1 (primarily for debugging and profiling)

[3] There was a relatively small number of JNI methods that did not either have a Java interface or C/C++ implementation. These unusable methods were excluded from our survey.

[4] Because of the limitation of the IMSI taint source as discussed in Section 8, we disabled the IMSI taint source for experiments. Nonetheless, TaintDroid's flag of the ICC-ID and the phone number led us to find the IMSI contained in the same payload.

[5] Regardless of the string separation, the MCC and MNC are identifiers that warrant taint sources.

StarTrack Next Generation: A Scalable Infrastructure for Track-Based Applications

Maya Haridasan, Iqbal Mohomed, Doug Terry, Chandramohan A. Thekkath, and Li Zhang

Microsoft Research Silicon Valley

Abstract

StarTrack was the first service designed to manage tracks of GPS location coordinates obtained from mobile devices and to facilitate the construction of track-based applications. Our early attempts to build practical applications on StarTrack revealed substantial efficiency and scalability problems, including frequent client-server roundtrips, unnecessary data transfers, costly similarity comparisons involving thousands of tracks, and poor fault-tolerance. To remedy these limitations, we revised the overall system architecture, API, and implementation. The API was extended to operate on collections of tracks rather than individual tracks, delay query execution, and permit caching of query results. New data structures, namely track trees, were introduced to speed the common operation of searching for similar tracks. Map matching algorithms were adopted to convert each track into a more compact and canonical sequence of road segments. And the underlying track database was partitioned and replicated among multiple servers. Altogether, these changes not only simplified the construction of track-based applications, which we confirmed by building applications using our new API, but also resulted in considerable performance gains. Measurements of similarity queries, for example, show two to three orders of magnitude improvement in query times.

1 Introduction

The easy availability of function-rich mobile devices has fueled significant interest in the "mobile internet", where mobile devices access internet-based services and web applications. Mobile devices that can determine their own physical location are adding to this trend by facilitating the development of diverse location-based services. In addition to individual coordinates, "tracks" — time-ordered sequences of GPS locations recorded by mobile devices — enable many location-oriented applications, varying from personal applications such as trip planning and health monitoring, to social applications such as ride-sharing and urban sensing.

StarTrack, introduced in an earlier paper, was the first service designed to manage tracks from mobile devices and to facilitate the construction of track-based applications [3]. That paper was primarily focussed on identifying a rich class of interesting personal and social applications that exploited histories of tracks; not much attention was paid to implementing the service at scale or building applications. Indeed, the entire implementation relied heavily on the services of a single database server with a thin software veneer providing an API. No applications were built using this API. Our first attempt to build realistic applications using this system revealed many shortcomings: principally inadequate performance, scalability, and fault-tolerance. Some of these, e.g. fault-tolerance, arose out of inadequate system structure in the original implementation. But by far most of the shortcomings arose out of a mismatch between the API provided by the system and what was required by applications. Specifically, several functions that were necessary for applications were either missing in the API or needed to be synthesized from lower-level primitives of the API. This mismatch led to costly and unnecessary client-server communication and data transfer. In addition to these deficiencies, our original system implemented common operations inefficiently (e.g. track comparisons).

This paper describes how the design and implementation of StarTrack have evolved non-trivially to address real-world issues of dealing with tracks. Our experience with track-based applications is admittedly limited. We do not claim our API is universal or fundamental in any sense; it will undoubtedly evolve as we encounter new classes of applications that we have not anticipated. Nonetheless, we believe our work and experience to date will be beneficial to researchers and practitioners in this rapidly growing field.

In general, we found managing and providing semantically rich operations on tracks to be surprisingly difficult. Track queries are complex because they involve geographic and similarity constraints, and a naive solution requiring expensive evaluation of these constraints does not scale to real-world online demand.

The main insight we use in tackling the complexity of tracks is to recognize that tracks tend to be repetitive. Repetitiveness arises from two distinct sources. An individual tends to follow substantially similar routes in his day-to-day life. This intuition is supported scientifically by a recent study in Science [23]. Second, the vast majority of tracks are collected on roads and highways, again leading to significant overlap in tracks even if they are from different users.

This insight permeates all parts of our revamped StarTrack infrastructure. We made several changes to our system. In some cases, we needed new techniques and data structures; in other cases, we used more established techniques, but synthesized in novel ways, to support a new class of track-based applications efficiently.

The changes to our system fall into four broad areas:

API Changes. All operations in our original API dealt with individual tracks, often causing entire sets of tracks to be moved repeatedly between the service and applications. StarTrack currently supports a "track collection", representing a set of tracks. Several functions in the API now operate on and return results as track collections. This change had several benefits. Apart from the obvious ease of programming, it afforded StarTrack opportunities to optimize the performance of specific operations through delayed and partial evaluation of these collections. Caching of both full and partial results also became possible.

Changes in Track Representation. We quickly discovered dealing with "raw" tracks by themselves to be inefficient. We now use a "canonical" representation for tracks, where tracks are represented as a sequence of points drawn from a fixed set, such as road intersections. Canonicalization benefits many aspects of the system. It reduces the computational costs of track comparison while improving its accuracy. As a consequence of improved accuracy, we are able to group a user's similar tracks more effectively and maintain a small set of representative tracks that captures the essentials of a large set of tracks. Many applications only need to operate on the set of representative tracks, leading to significantly fewer operations, better caching of data, and consequently, better performance.

Changes to On-Disk and In-Memory Data Structures. The original StarTrack API was implemented as a thin veneer on top of a geospatial database system. While simplifying the implementation, this resulted in poor performance for many operators. The changes in the API and canonicalization described above allowed us to build specialized in-memory data structures to augment the database tables. Operations that had low performance are now optimized by using in-memory quadtrees or a novel structure called a *track tree* described in Section 3.3. In addition to these in-memory data structures, we reorganized the database layout to include a table of representative tracks for each user (as mentioned above) and other tables that aid in handling operations with geographical constraints.

Structural Changes. Our original prototype consisted of a single server process that stored tracks in a centralized database and implemented an API to access these tracks. This single server implementation clearly did not scale to a large number of tracks or provide fault-tolerance. In the new system, a set of StarTrack server machines connects to another set of database servers. Applications use a StarTrack clerk, which implements the API and makes remote procedure calls (RPCs) to the StarTrack servers as necessary. It also deals with retrying requests on server failures, and balances RPC requests amongst servers.

We detail our changes further in the rest of the paper (Sections 2–4), describe two scalable, robust, and efficient applications they enabled us to build (Section 5) and summarize their performance impact (Section 6).

2 Application Programming Interface

The interface exported by the StarTrack service has undergone multiple revisions based on our experience building realistic applications. This section describes the key elements of the new application programming interface; space restrictions prevent us from describing the complete API.

2.1 Track Collections

The new StarTrack interface supports the notion of a *track collection*, an abstract grouping of tracks, where the application supplies the criteria for grouping. Track collections can, in turn, participate in other StarTrack operations. All non-trivial operations in the StarTrack API take a track collection as an argument.

Track collections have two significant advantages:

Implementation Efficiency. They allow the server to treat the set of tracks that are repeatedly accessed together as a single entity for the purposes of caching. They also allow the server to construct specialized data structures that operate exclusively on these tracks, making these operations more efficient. Furthermore, by having applications and the service refer to a potentially

large collection of track identifiers by a single identifier, we reduce the communication costs of transmitting the identities of individual tracks between them.

Programming Convenience. Applications often want to constrain operations to tracks that belong to a particular community or cohort. For example, a social application might wish to operate on the tracks of a user and his group of friends. Track collections allow such an application to create an aggregation of the tracks in which it is interested and enable it to operate on such groups more conveniently.

Track collections are created by using the *MakeCollection* procedure (see API Fragment 2.1). *MakeCollection* takes as its first argument a set of criteria to select a group of tracks from all tracks in the system. Individual criteria can be composed out of three elements: *geographic*, *time*, *user*. The first two elements have fairly simple semantics: a geographic element is specified by a physical geographical region and a time element is specified by a time interval. The user element consists of two subfields: a unique identifier that specifies the user and a string field that specifies an XPATH query. The query is applied to the user metadata that is stored in the track by the application.

```
TrackCollxn MakeCollection(GrpCriteria[] gCrit,
                           bool unique);
```

API fragment 2.1: Operation to create a track collection.

The second argument is a boolean that indicates whether the system should return only "unique" tracks. Two canonical tracks are considered unique if their starting points (as well as ending points) are "close" to each other, and their paths are highly "similar" to each other. Similarity is more precisely defined below when we discuss the *GetSimilarTracks* function. Parameters that decide if the start/end points are "close" to one another and if tracks are highly similar are defined by the infrastructure. These are described further in Section 4.1.

We provide applications the option to specify the unique flag for two reasons. People tend to travel the same routes habitually, leading to multiple highly similar tracks that only differ in time. Meanwhile, many applications are only interested in distinct routes without requiring knowledge of the precise times at which the route was traveled. These applications greatly benefit from using *MakeCollection* with the `unique` flag set to true since it significantly reduces the number of tracks in the returned collection. If instead an application needs per track information, for instance, if it needs to know how fast the user travels on a particular road segment, setting `unique` to false will retrieve all the relevant tracks with detailed information.

Two simple code segments calling *MakeCollection* are shown in Examples 2.1 and 2.2. The first example collects the tracks of user Uriah between 8AM and 10AM. The second shows how metadata information is used to create a track collection of all employees of an organization.

Example 2.1 Uriah's tracks between 8AM and 10AM.

```
GrpCriteria[] gCrit = new GrpCriteria[2];
UserCriteria uc = new UserCriteria();
uc.Username = "Uriah";
TimeCriteria tc = new TimeCriteria();
tc.StartHour = 8;  tc.EndHour = 10;
gCrit[0] = uc; gCrit[1] = tc;
TrackCollxn tcUriah;
tcUriah = MakeCollection(gCrit, false);
```

Example 2.2 Tracks of all employees of the Wickfield corporation. The metadata string is an XPATH query, shown here in simplified syntax for formatting reasons.

```
GrpCriteria[] gCrit = new GrpCriteria[1];
UserCriteria uc = new UserCriteria();
uc.metadata = ``Employer = Wickfield'';
gCrit[0] = uc;
TrackCollxn tcWField;
tcWField = MakeCollection(gCrit, true);
```

2.2 Manipulating Tracks

Tracks can be manipulated in several ways; we describe a few representative operations. We have chosen these because they embody the most significant changes we made to the original prototype. Other operations are essentially unchanged from our previous API.

JoinTrkCollections takes two or more track collections and creates a new track collection that is the union of all the constituent tracks. The second argument allows the resulting track collection to retain only unique tracks. *SortTracks* takes a track collection and orders the constituent tracks in the collection according to one of a set of predefined attributes. Examples of attributes we have implemented are `LENGTH` and `FREQ`, which refer to the length of the track and its frequency of occurrence within that track collection.

Many track-based applications need to determine whether tracks are similar to one another. Given two tracks, we define track similarity as the ratio of the length of all the segments that are common to both of them divided by the length of the union of all segments present in either of them (Figure 1(a)). *GetSimilarTracks* is given

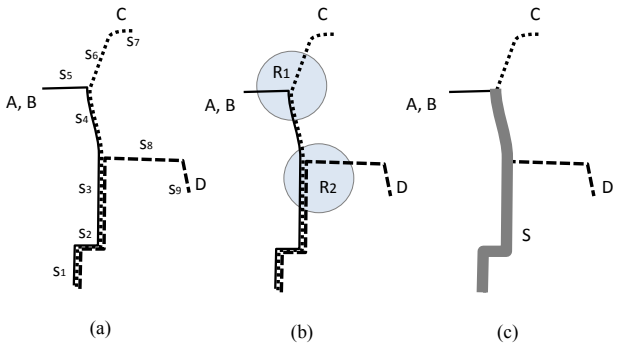


Figure 1: (a) the similarity between tracks A and B is 1 and between A and D is $(l_1 + l_2 + l_3)/(l_1 + l_2 + l_3 + l_4 + l_5 + l_8 + l_9)$, where l_i is the length of segment s_i; (b) A,B,C are the tracks that pass by the areas R_1 and R_2; (c) S is the common segment of A,B,C,D with frequency threshold set to 0.6.

```
TrackCollxn JoinTrkCollections(TrkCollxn tCs[],
                        bool unique);

TrackCollxn SortTracks(TrkCollxn tC,
                       SortAttribute attr);

TrackCollxn GetSimilarTracks(TrkCollxn tC,
              Trk refTrk, float simThresh);

TrackCollxn GetPassByTracks(TrkCollxn tC,
           Area[] areas);

TrackCollxn GetCommonSegments(TrkCollxn tC,
           float freqThresh);
```

API fragment 2.2: Operations to manipulate a track collection.

```
int GetTrackCount(TrkCollxn tC);
Track[] GetTracks(TrkCollxn tC, int start,
                            int count);
```

API fragment 2.3: Retrieval operations on a track collection.

a track collection and a reference track and selects from within the collection all tracks that are similar to the reference track. The returned track collection is sorted by similarity. The degree of similarity is controlled by the third parameter.

Track-based applications can find tracks that pass within close proximity of a location by calling *GetPassByTracks*. *GetPassByTracks* is given a track collection and an array of `Area` objects and returns all tracks in the collection that pass through all the areas (Figure 1(b)).

GetCommonSegments takes a track collection and a frequency threshold and returns the road segments shared by at least that fraction of the tracks in the collection. These road segments are merged into the smallest number of contiguous routes possible (see Figure 1(c)). This operation is useful for the application to retrieve a succinct summary of a potentially large set of tracks.

Tracks within a `TrackCollxn` object can be retrieved via the following two functions (See API Fragment 2.3). *GetTrackCount* returns the number of tracks in a track collection, and *GetTracks* returns `count` tracks beginning at the `start` location within a track collection.

3 StarTrack Server Design

This section describes three changes to the StarTrack server design that we consider most significant.

3.1 Canonicalization of Tracks

In our first implementation, we stored users' latitude and longitude coordinates directly in the system. While this design choice was intuitive and useful in some circumstances, it was problematic in many others. Recall that coordinates are samples of a path taken by a user. The same path taken by different users may be sampled at different points. Also, sampling is inherently error-prone due to limitations in current localization techniques [8]. For these reasons, two identical paths can lead to widely different sampled coordinates, making it difficult to classify them as equal. In the new system, we "canonicalize" paths to eliminate spurious variability in the sampled coordinates. In this context, canonicalization means that we convert a path to another path that only passes through a set of "standard" points drawn from a (large) fixed set. We refer to the portion of the path between two such points as a segment.

There are several methods to canonicalize tracks. One intuitive way is to overlay a fixed grid on the geographic region and to map each coordinate to a grid intersection point. A variation on this technique is to pick a suitably weighted interior point within the grid instead of a corner.

A fundamental shortcoming of approaches based on a fixed grid is that the grid is artificially created and does not adapt to users' tracks. Grids may be too fine-grained, in which case canonicalization provides no benefits, or too coarse-grained, in which case important features of tracks are lost.

Instead of using an artificial grid, we can often use the more natural and adaptive grid imposed by streets and highways. Canonicalizing based on street maps is called *map matching* and is desirable in cases where roadmaps of the region exist. A track after canonicalization is mapped to a path in the roadmap. A path consists of one or more street segments and is stored as a sequence

of the endpoints of the segment(s). StarTrack uses a map matching approach using hidden Markov models designed by Krumm *et al.* [17, 20].

The performance of canonicalization is dependent on three factors: the sampling rate of a track (i.e., the number of GPS points in the track), the length of the track, and the amount of GPS noise introduced into the samples. In our system, canonicalization is done offline as a pre-processing step. Since the performance of canonicalization is not that critical in our system, we do not present detailed results. With some performance tuning, StarTrack can canonicalize a track with average trip length of about 20 km and 400 GPS samples in under 250 ms.

Canonicalization has two key advantages that translate into performance savings. First, StarTrack can compare two segments for equality without using expensive geographic constraints. Equality of segments is used within the inner loop of the procedure that finds similar tracks, which in turn is a very common operation in applications. Second, canonicalization tends to create larger numbers of identical segments. This often allows us to access and manipulate a single representative segment rather than dealing with individual segments. It also allows StarTrack to identify duplicate tracks more accurately and reduces the number of tracks it needs to process for various operations.

Canonicalization based on road networks is appropriate for regions that have a mature road network and a stable map. When road networks are not available, we may utilize technologies for constructing road maps from user tracks [5, 7].

3.2 Delayed Evaluation

We found that applications typically make several API calls to narrow down the set of tracks they want to retrieve. Our implementation of the API therefore delays the evaluation of the tracks in a track collection until one of the two retrieval functions in API Fragment 2.3 is called. This technique saves multiple roundtrips between the StarTrack clerk and servers. Furthermore, it allows the StarTrack server flexibility in the queries it issues to the database and in the choice of data structures it builds for different retrieval operations.

When a client invokes a *MakeCollection* operation, the client-side stub marshals an efficient description of the call arguments and a small integer representing the procedure name. We call the resulting structure a *descriptor*. The stub sends the descriptor to the server, which stamps it with the current time to capture the database contents at that instant and returns it.[†] We require that the timestamp be in the past with respect to the time on the database server. Assuming that tracks are not deleted from the system, this guarantees that multiple evaluations of a track collection will always return the same set of tracks.

Operations such as *JoinTrkCollections*, *GetPopularTracks*, *GetSimilarTracks*, and *GetPassByTracks* create compositions of these descriptors (at the client stub) with no communication to the server and no additional timestamps. We refer to these compositions as *compound descriptors*. These are organized as a tree, with the leaves being a simple timestamped descriptor.

Notice that all descriptors (compound or otherwise) contain information about the invoked function and the arguments, which together can be used to construct a track collection. In this sense they can be viewed as a closure [18] or as a specialized form of a logical view from the database literature [9].

Our use of timestamped descriptors is a tradeoff between efficiency and freshness. Timestamps imply that the application sees data as it existed in the database at a particular point in time, not necessarily the latest data. It allows the StarTrack server to cache the contents of the database in an in-memory data structure, or discard it at will and reevaluate it later, while providing easy to understand and consistent semantics to the application. It also allows a client to present the descriptor to a different StarTrack server if needed for load-balancing reasons or if the original server crashes. Re-evaluating a descriptor is guaranteed to yield the same result anywhere in the system because the operations are deterministic, and the timestamp acts as a snapshot of the database (provided that tracks are not deleted from the system). If freshness is more important for an application, it can recreate the track collection as often as needed.

The evaluation of a descriptor yields different types of in-memory data structures. For example, the evaluation of a descriptor constructed by *GetSimilarTracks* may (but need not) create a data structure called a track tree. A descriptor created by *GetPassByTracks* can result in a quad-tree [10]. The results of evaluating other descriptors are typically stored as a simple set of tracks.

3.3 Track Tree

In our experience, when two tracks overlap, they usually do so on one or very few contiguous segments. We exploit this property to build a hierarchical data structure called a *track tree*, which is used to speed up the retrieval of similar tracks.

Each road segment is represented as a leaf node in a track tree. For each leaf node, the track tree records all tracks that contain that particular segment. Once all the segments in a track collection are stored as leaf nodes, pairs of nodes that refer to geographically adjacent segments are considered for merging to form interior nodes

[†]There are well-known ways to avoid this RPC call, but we have chosen not to implement them for simplicity.

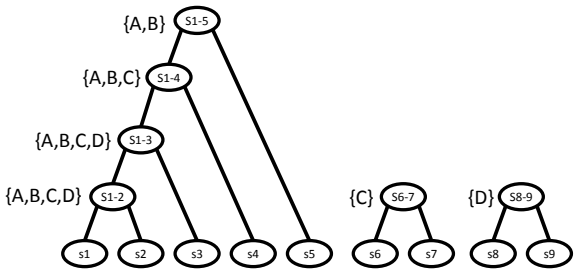


Figure 2: The track tree of the set of four tracks shown in Figure 1(a). Each node, except for leaf nodes, is annotated with the set of tracks that contain it.

of the tree. Whenever there is choice of pairs of nodes to merge, the pair that has the highest number of tracks in common is picked. This process is continued iteratively up the tree. When merging two nodes, all tracks belonging to both children nodes are included in the parent node as well. By this construction, each node in the track tree represents a contiguous sequence of road segments. In addition, the segment is more likely to be shared by multiple tracks.

Figure 2 shows the track tree for the sample four tracks in Figure 1(a). As shown in Figure 1(a), tracks A and B are identical and consist of segments S1, S2, S3, S4, and S5. Tracks C and D share common segments with A and B. Segments shared by larger numbers of tracks are favored when merging nodes, which explains why segments S1 to S3 are merged together, instead of other combinations, such as S2 to S4. Using this tree, tracks A and B can be described by one single node (S1-5), and tracks C and D can be described by two nodes each: Track C by S1-4 and S6-7 and Track D by S1-3 and S8-9.

Track trees are used to accelerate several API operations. In *GetCommonSegments*, after we identify the road segments shared by sufficiently many tracks, as indicated by the given threshold, we use a track tree to organize them into a small number of contiguous tracks. This is done by merging up in the tree those nodes corresponding to these road segments. Given the way a track tree is constructed, this usually results in a small number of nodes, corresponding to a small number of contiguous tracks.

Another API operation enabled by a track tree is *GetSimilarTracks*. Implementing this function as a database operation is inefficient because there is little match between our similarity semantics and the primitives supported by spatial databases.

With a track tree, StarTrack can quickly find a set of tracks with a given degree of similarity to a specific track T (See Code Segment 3.1). First, StarTrack identifies the set of all nodes (interior and leaf) covered by T. In order to do this, T is initially broken into smaller segments. StarTrack then identifies the leaf nodes in the track tree that correspond to these segments. Next, it identifies pairs of adjacent nodes that have a common parent node, includes the parent into the set, and iterates until no such parent exists. These steps are encapsulated in the function `Map`.

The *GetSimilarTracks* operator then sorts the nodes in T by decreasing order of length. It sequentially scans each node, examining the set of tracks containing it, and outputs tracks that are at least `simThresh` similar to the query track. This process stops when it has found sufficiently many tracks as defined by the `maxCount` parameter, or when it has examined sufficiently many tracks. Recall that the client supplies the `simThresh` parameter (as part of the *GetSimilarTracks* call), as well as the `maxCount` parameter (as part of the *GetTracks* invocation, which triggers the evaluation of the descriptor). This process will not produce any false positives (i.e., tracks that purport to be similar but are not), but it could miss some highly similar tracks. The percentage of such misses is quite small when the similarity threshold is reasonably high, as our experimental results show (see Figure 7(c) in Section 6).

Code segment 3.1 Pseudo-code for implementing `GetSimilarTracks` using *tracktree*.

```
Track[] GetSimilarTracks(TrackTree trackTree,
  Track T, double simThresh, int maxCount)
{
  TrackTreeNode[] nodes = trackTree.Map(T);

  SortByDescLength(nodes);

  SortedList<Track> results; int examined = 0;
  foreach(node in nodes) {
    foreach(candidate in node.tracks) {
      if(T.Similarity(candidate)>=simThresh)
        results.Add(candidate);
      examined++;
      if((results.Count>=maxCount)||
         (examined>=6*maxCount))
        return results;
    }
  }
  return results;
}
```

Similar to other in-memory data structures in StarTrack, a track tree is cached in memory until evicted under the caching policy: LRU in our implementation. Since track collections are immutable, we do not update data structures during their life time. However, the track tree structure allows for efficient insertion of new tracks, and whenever a track collection is created by building upon an existing track collection, an existing underlying track tree may be copied and updated.

4 Storage Platform Design

As previously described, we build and maintain in-memory data-structures at the StarTrack servers, and use a different set of database servers to store data persistently. StarTrack always checks if tracks can be found in the in-memory data-structures before fetching them from the database.

StarTrack uses Microsoft's SQL Server 2008, which supports the notion of geospatial objects as a fundamental data type. Data is partitioned across multiple machines, and partitions are replicated using chained declustering [12], which provides the necessary scaling properties as well as automatic dynamic load-balancing and fault-tolerance.

4.1 Database Tables

The principal on-disk data structure consists of 5 tables stored in SQL Server.

User Table. This consists of a set of records for each user containing a unique system-assigned user identifier and other personal information.

Track Table. Every track is assigned a unique identifier, consists of a set of time-stamped latitude and longitude coordinates, and is stored in a single row in the table. Both the raw and the canonical versions of tracks are stored in the same table.

Representative Track Table. This table maintains a set of representative tracks per user and allows StarTrack to often avoid searching the larger Track Table. Each record stores information related to a single representative track: the canonical coordinates, the owner, and a count of how many actual instances of this representative track exist in the Track Table. Upon insertion of a track into the Track Table, StarTrack checks if there exists a representative track that matches the new track. If so, the new track is not inserted into the Representative Track Table, but the count of the matching representative track is incremented. The count serves as indication of the popularity of a given representative track and is used by StarTrack operations for ranking purposes.

Two tracks are considered as matching if their start points are within 100 m of each other, if their end points are within 100 m of each other, and if the tracks are at least 90% similar. The choice of these parameters is fixed by the infrastructure and cannot be changed by individual applications. It is based on expected errors in GPS measurements, as well as cost/benefit tradeoffs, and is not as Procrustean as one might imagine. The values chosen determine the size of the Representative Track table — high start/end point buffer values and low track similarity values result in a smaller table of unique tracks, but applications may lose the ability to discriminate between tracks. The size of the table, in turn, affects the speed of many functions in the API that must access that table.

Coordinate Table. During the map matching process, the set of coordinates in a path is drawn from a finite list of points, which depends on the particulars of the map data used for canonicalization of tracks. Each record in this table maps a location identifier to a pair of coordinates. This particular table is immutable, replicated on each database server, and not partitioned.

Coordinate to Track Table. This table maps coordinates to tracks that go through them. We use it to speed up the location of tracks that pass through certain geographic boundaries.

StarTrack allows three types of criteria in fetching tracks from the database: user, time, and geographic region. Region-based queries may be performed by leveraging the geospatial functions provided by modern database systems, which support specialized indexing schemes. Such systems must be used with care because costs are still significant when indexing large numbers of complex geospatial objects such as tracks.

In the original StarTrack implementation, we used the geospatial primitives of the database to treat each track as a separate object and created a geospatial index over all such objects. Now, we maintain a geospatial index on the Coordinate Table alone, thereby reducing the number of objects on which the geospatial index is maintained. We use this index to find all locations that match a given geographic query. We then use the Coordinate To Track Table to look up all tracks that go through these locations. This is feasible precisely because of the canonicalization pre-processing step.

The Coordinate Table and its geospatial index are maintained by the database server and portions of them may be cached in memory. We present a comparison of the original and new approaches in Section 6.2 (Figure 3). If necessary we can further speed up our design by not storing the Coordinate Table in the database server and can instead store it in memory and index it using an in-memory quad-tree.

4.2 Database Server Organization

The tables mentioned above are partitioned across multiple database servers. Based on StarTrack's search criteria options, we considered two partitioning schemes: by geography and by user identifier.

We decided not to partition by geography, since over time it would lead to increasing numbers of tracks that span geographic regions, therefore having to span servers.

We opted for partitioning data by user identifier, keeping all data referring to a single user in a single database server. This organization allows user-constrained queries to be sent to a single database server, while requiring geographic queries to be sent to all database servers.

Data is mirrored in the system. Each database server acts as the primary for one partition of each table, and as the mirror (or secondary) for its neighbors' partitions. A primary database server processes read and write requests from clients, while a mirror server only handles read requests.

StarTrack servers are clients of the database servers, and evenly distribute reads amongst the replicas. When a database server fails, the server that mirrors the partitions on the failed server takes over as primary for the partitions. The StarTrack servers direct write traffic to the new servers and in addition, distribute the read requests uniformly among all the replicas using chained declustering, as described by others [12, 19].

5 Applications

We explored scenarios where a single user's data can be used to personalize her experience based on her habitual tracks, for applications such as personalized advertising, recommendation systems, and health monitoring. On the other end of the spectrum, social applications, where the set of tracks from a group of friends or even a broader community are used, may help provide enhanced services to users. Examples include those related to urban sensing, collaboration, discovery of new areas, and shared experiences.

To illustrate the usefulness and evaluate the performance of StarTrack services, we describe two of the applications we built.

While both applications were non-trivial to write, the use of our API significantly simplified their construction. In fact, the application logic in both examples is succinctly captured in a few code snippets. Our general experience is that StarTrack provides an intuitive, flexible, and efficient way to program track-based applications.

5.1 Ride-Sharing Service

Ride-sharing has long held the promise of reducing energy consumption. Transit departments in many major metropolitan areas now offer on-line ride-sharing services or portals (see for example, King County Metro Ride [15]). One challenge in building an effective ride-sharing service is to discover ride-share partners who travel on similar routes.

With StarTrack, these ride-matching services are easily built. The service can build a TrackCollection for the employees of the same company or for a person's social network, or for a group of people who have subscribed to a transit service. Code Segment 5.1 constructs a track collection for a community of users. Code Segment 5.2 identifies potential ride-sharing partners based on the similarity of their travel patterns.

Code segment 5.1 Set up a community's regularly traversed tracks where the community is defined through supplied `SearchCriteria`.

```
TrackCollxn getCommunityTracks(SearchCriteria sc,
                          int count)
{
    TrackCollxn tc = MakeCollection(sc, true);
    return Take(SortTracks(tc, FREQ), count);
}
```

Code segment 5.2 Find ride-share candidates with similar travel patterns. `findOwners` is a client-side function that takes a set of tracks and returns the list of users who own them.

```
List getRideShareCandidates
    (TrackCollxn communityTC, string username)
{

    UserCriteria uc = new UserCriteria();
    uc.Username = username;
    TrackCollxn userTC =
        MakeCollection(uc, true);
    Track[] popularTracks =
        GetTracks(SortTracks(userTC, FREQ),
                                0, 10);
    List<TrackCollxn> similarTC;
    foreach(Track track in popularTracks) {
        TrackCollxn tc = GetSimilarTracks(
                    communityTC, track, 0.7);
        similarTC.Add(tc);
    }
    Track[] similarTracks =
        GetTracks(JoinTrackCollections(similarTC)
                          0, 100);
    return findOwners(similarTracks);
}
```

Another usage scenario is when a user needs a ride between two specific locations. This can be done easily by calling *GetPassbyTracks*.

It is important to note that the ride-sharing service based on StarTrack offers more flexibility than conventional services. For instance, since a rider's entire route is known, rather than just his start and destination, it allows the service more latitude in arranging pick-ups and drop-offs along the route.

5.2 Personalized Driving Directions

Current navigation systems and online map services provide detailed turn-by-turn driving directions. Because

StarTrack knows what routes a person has taken in the past, as well as how recently and how frequently, an application could easily use StarTrack to provide personalized driving directions.

For example, instead of providing detailed turn-by-turn instructions on how to get to the freeway from the person's house, the directions might simply say "Get on Highway 101 heading south" and then provide detailed directions from that point.

Code segment 5.3 Construct a user's familiar segments.

```
TrackCollxn getFamiliarSegments(string username)
{
    UserCriteria uc = new UserCriteria();
    uc.Username = username;
    TrackCollxn uTC = MakeCollection(uc, true);
    // Pick the 10 most frequently occurring
    // tracks.
    TrackCollxn pplrTC =
                Take(SortTracks(uTC, FREQ), 10);
    TrackCollxn familiarTC =
                GetCommonSegments(pplrTC, 0.2);
    return familiarTC;
}
```

The application we built uses the Bing Map service and the StarTrack infrastructure. A user inputs start and destination locations, and the application uses Bing to get turn-by-turn directions for that route. Next, the application uses StarTrack to obtain the set of "familiar segments" for that user, as shown in Code Segment 5.3.

Having obtained the familiar segments for the user, the application identifies portions of the route returned by Bing that overlap with the familiar segments and uses the result to prepare personalized driving directions (we omit further description of these steps given that they are performed locally by the application and do not involve calls to StarTrack).

6 Evaluation

This section evaluates the performance of the StarTrack service. To study the system at scale, we used synthetically generated tracks. We also ran experiments with actual tracks collected by users of GPS-equipped mobile devices, but omit the results since they are similar to those performed with synthetic tracks, and given that we only have a limited number of real tracks.

We focus on the costs of executing track operations that involve (a) geographic constraints and (b) comparisons of tracks. These operations are the most difficult to build efficiently, and are also among the most commonly occurring in the track-based applications that we built. We also report on the performance of two applications.

Our experiments were all conducted on 2.6 GHz AMD Opteron quad-core processors with 16 GB memory, running Windows Server 2003.

6.1 Synthetic Tracks

We generated synthetic tracks based on the salient features observed in a dataset of approximately 16,000 real tracks followed by 252 users over 2-week periods in Seattle, WA [16]. In our model, each person has fixed locations for home and workplace, and a number of "errand" locations that represent places they go less frequently. On weekdays, a person travels between the assigned home and work locations during the common morning and evening commute hours. Sporadically on weekdays and more often on weekends, a person carries out a number of errands.

After choosing the start and end locations for each trip, we calculate the shortest path as well as its duration between these points on a graph of road networks. We then sample and perturb each path to simulate noise in the sampling and localization of the data and treat the resulting points as a track.

Our early experiments indicated that some features of tracks have a pronounced effect on performance while others do not. Specifically, performance is affected by the following:

- Number of tracks. The larger the number of tracks, the greater the computational and storage overhead.
- Length of tracks. The number of points in a track has an impact on performance. Assuming tracks are canonicalized, the number of points is proportional to the length of the tracks.
- Covered region. The region over which the tracks are generated has an impact on track density (i.e., number of tracks that pass through a unit area). As track density increases, the computational burden imposed on our algorithms increases. For example, the same geographic query returns more results and therefore incurs more computational cost when the density of tracks is higher.

We devised our model to allow us to control these key features. Our belief is that, at least for the purpose of performance evaluation, any model that allowed these features of tracks to be varied would be adequate.

For our scalability experiments, we generated synthetic tracks for a 3-month period and 18,000 users in Santa Clara County. This resulted in a total of over 4.5 million tracks. On average, each track is 20 km long and contains 400 GPS samples that yield on average 163 points after canonicalization.

6.2 Performance of Geographic Queries

One of StarTrack's most important operations is querying based on geographic constraints. Some of these operations require a round-trip to the database server, while others can be optimized by an in-memory cache. In our API, geographic queries show up in two forms. First, in *MakeCollection* an application can specify a geographic region constraint. Second, *GetPassByTracks* allows an application to select those tracks in a track collection that pass within specific areas. The first query involves retrieving tracks from a database, while the second involves retrieving tracks from a pre-computed track collection, which can be sped up in memory.

Geographic queries to the database. Although we do not focus on studying the performance of the spatial features of the database, we investigate how best to use them to improve simple geographic queries used to pre-filter tracks brought into memory.

We compared two ways to store tracks and construct the necessary indices. In the first approach, used in our original prototype, we treat each track as a separate geospatial object and create a spatial index over all tracks. This index is used to retrieve all the tracks intersecting the query region. The second approach, used by StarTrack, involves the use of two additional tables, the Coordinate Table and the Coordinate to Track Table, as described in Section 4.1. In this approach, a spatial index is built only on the Coordinate Table.

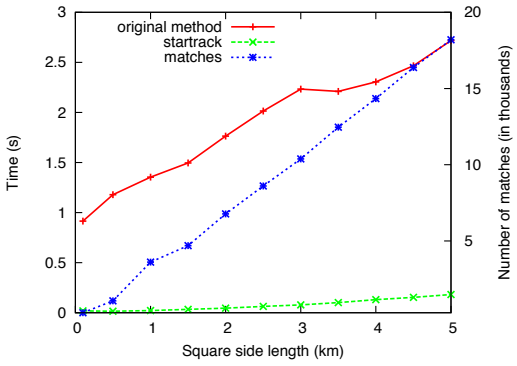


Figure 3: Query time with and without the Coordinate Table when searching for tracks that intersect square regions of increasing side lengths. Secondary y-axis shows average number of tracks matched.

Figure 3 presents the query time for both approaches when we vary the area of the query region on a set of 100,000 tracks. It also shows the average number of matched tracks on the secondary y-axis. Isolating the need to execute geographic queries to a small set of distinct points through the use of the Coordinate Table leads to significant performance benefits. This enhancement is only possible due to track canonicalization.

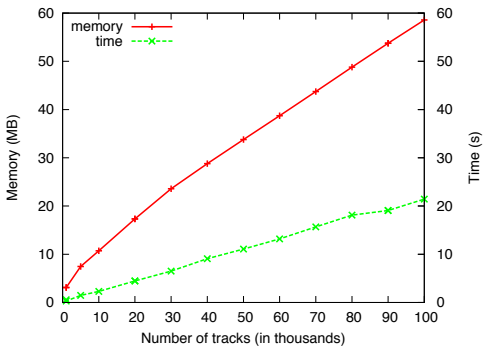


Figure 4: Memory usage and construction time of the quad-tree for different sizes of tracks.

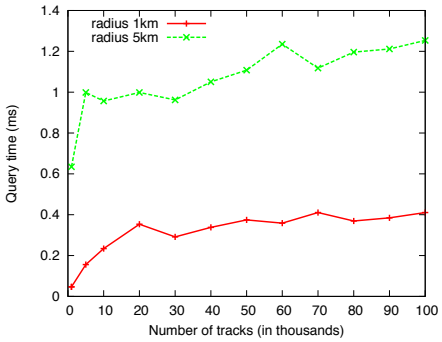
Geographic queries to in-memory data structure. Recall from Section 3.2 that the evaluation of a *GetPassByTracks* operation triggers the construction of an in-memory quad-tree, in the expectation that the data will be repeatedly accessed in the future. Canonicalization tends to lower the number of unique coordinates in tracks, speeding up the construction time for quad-trees, as well as the execution time of subsequent requests against it. Figure 4 shows the cost of constructing a quad-tree. Building the quad-tree itself requires little space and time since the number of unique coordinates is small and levels off when the tracks cover a large region. Both the memory and time needed are linear in the number of tracks, and are mostly spent on building an index from coordinates to their containing tracks.

Figure 5 presents the time to query a quad-tree with varying numbers of tracks and region sizes. In all cases, the query time is very low. For example, it takes about 1 ms for a region with a 5 km radius on 100,000 tracks. The query time is fairly insensitive to the number of tracks because the structure of the quad-tree is determined by the unique coordinates. On the other hand, the size of the query region affects the times since it determines the number of quad-tree cells to be visited.

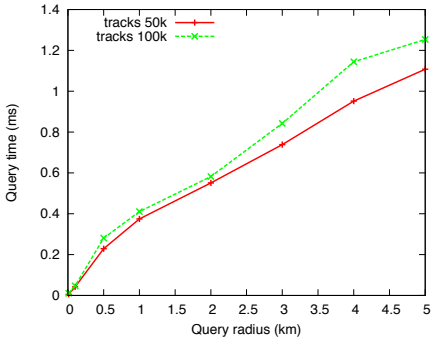
6.3 Performance of Track Comparisons

A common query in track based applications is to retrieve tracks based on similarity. Typically, an application has a track collection and a "query" track and needs to find tracks in the set that are most similar to the query track.

We compare the performance of our technique using a track tree to three alternative methods for ranking tracks based on similarity: (1) *Bruteforce*: The brute-force method compares the query track against every track in the collection and returns those with similarity



(a)



(b)

Figure 5: Time to query a quad-tree. (a) Query time for different numbers of tracks when size of the region is fixed to 1 and 5 km, respectively. (b) Query time for 50K and 100K tracks as the size of region is varied.

above a given threshold. For the bruteforce method, we assume all tracks are already in memory. (2) *In-memory filtering*: This method constructs an in-memory dictionary used to quickly look up tracks that contain any given point. For a given query track, we use this dictionary to identify all tracks that intersect it, after which we compute the similarity of each intersecting track to the query track, returning those above the threshold. (3) *Database filtering*: We store the set of tracks in the database, use a query to retrieve all tracks in the database that intersect the query track, and compute the similarity against the retrieved tracks.

We ran experiments with different numbers of tracks and queries with varying similarity thresholds.

Figure 6 shows the query time when using the various methods. The query time with the track tree method is dependent on the similarity threshold, unlike with the other three alternatives. In Figure 6, we present results for the track tree approach when the similarity threshold is 0.7 and 0.9. The experiments show that track trees lead to significantly more efficient queries when compared to the bruteforce method, achieving two to three orders of magnitude speedups. Although the in-memory filtering method performs better than the bruteforce method, it is still significantly slower while consuming high amounts of resources for constructing and storing the in-memory dictionary. The database filtering method presented the worst performance.

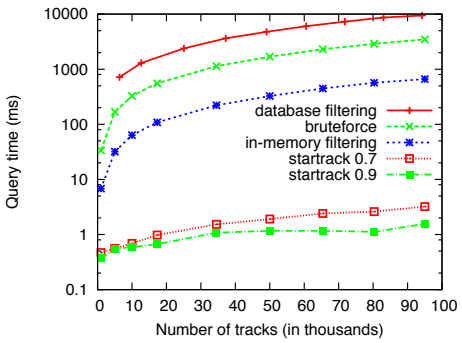
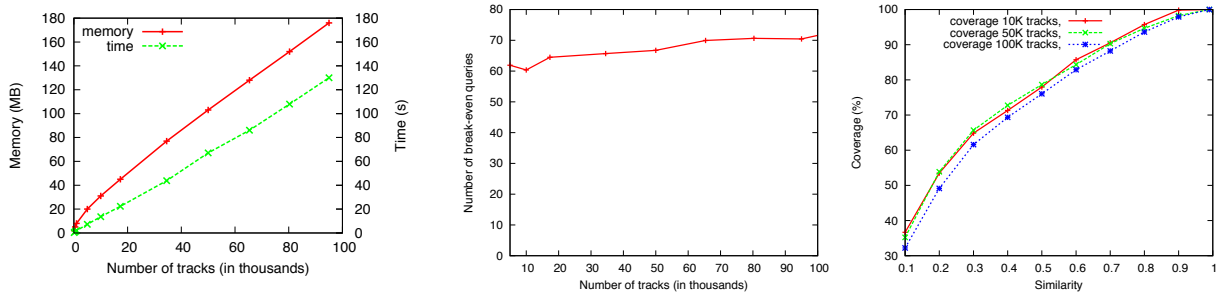


Figure 6: Query time comparison between StarTrack and three alternative methods. For StarTrack, results are shown for similarity thresholds of 0.7 and 0.9.

There is a cost associated with constructing a track tree that is at the heart of our technique. Figure 7(a) shows the memory usage and the time for constructing a track tree as a function of the number of tracks in the collection. Constructing a track tree takes linear space and slightly super-linear time as the height of the track tree grows logarithmically with the number of tracks. There is a tradeoff for using a track tree— it takes time to construct it, but once constructed, it leads to significantly optimized queries. From Figures 7(a) and 6, we calculate the "break-even" point, or the minimum number of queries such that the amortized query time using a track tree is lower than the query time of the bruteforce method. These break-even numbers are shown in Figure 7(b). As observed, the numbers grow slowly with the number of tracks, and are fairly small: below 80 for a track collection with up to 100,000 tracks.

One potential downside of the track tree approach is that while it is highly efficient at retrieving similar tracks and although it will never return tracks that do not satisfy the similarity threshold, it may not return all tracks above the given similarity threshold. Figure 7(c) shows the coverage of the track tree method. The graph shows the percentage of the expected tracks returned when using a track tree. We can see that the coverage increases for higher similarity thresholds. It returns over 90% of the tracks when similarity is above 0.7. We believe this is sufficient for typical applications, that are only interested in tracks with reasonably high similarity.



(a) (b) (c)

Figure 7: (a) Memory and processing time required for constructing a track tree. (b) Break-even number for use of a track tree. (c) Coverage of track tree approach as function of the similarity threshold (for 10K, 50K and 100K tracks).

6.4 Application Performance

We use the Ride-Sharing (RS) and Personalized Driving Directions (PDD) applications, presented in Section 5, to evaluate the overall performance of StarTrack. These two applications illustrate two different usage scenarios: RS creates a large track collection for repeated accesses while PDD creates many small per-user track collections.

We fixed the number of database servers to three and varied the number of StarTrack servers. To generate load on the servers, we ran multiple instances of these applications from a number of client machines.

6.4.1 Single StarTrack Server Experiments

The RS application identifies potential ride-sharing partners for a given user, and as presented in Code Segment 5.2, involves multiple calls to the StarTrack server. In our evaluation, we built a track collection with 50,000 unique tracks from which the application searches for similar tracks. We warmed up the server by constructing a track tree on the large set of tracks before sending it client requests. Figure 8(a) shows the response times for RS under varying request rates. Despite the more complex nature of the application, one StarTrack server is capable of satisfying 30 requests per second with a response rate of around 150 ms.

We ran experiments for the PDD application under two different types of load. In the first case, queries simulate users whose data has not been cached on the StarTrack server prior to the query. In the second case, we preload the cache with the in-memory data structures used to expedite the *GetCommonSegments* operation (*familiarTC* in Code Segment 5.3) invoked by the application.

Figures 8(b) and (c) plot the response times with varying request rates under the two types of loads. When the data is not cached, each server is capable of satisfying up to 30 requests per second without increasing the response time. The average response time prior to saturation is around 100 ms. The maximum server throughput increases to 270 requests per second and the response time falls to 60 ms when the data is previously cached on the server.

6.4.2 Scalability Experiments

For both applications, individual requests sent by the clients are entirely independent of one another. We tested StarTrack's scalability by running the PDD application on multiple StarTrack servers. For this experiment we used the non-cached version of PDD, with the goal of exercising load on the database.

In Figure 9 we present the maximum throughput that the system is able to achieve with a varying number of StarTrack servers. As expected, the system scales linearly with the number of servers. Since PDD only retrieves a small number of tracks for each user, this experiment did not saturate the database servers.

From these experiments, we estimate the resources needed to satisfy a given number of users for our tested applications. Three StarTrack servers can support a peak load of around 120 requests per second (without caching) or up to 780 (with caching). Without caching, this allows over 5 million queries uniformly distributed over a period of 12 hours, corresponding to an average of 5 queries per user given a population of 1 million users requesting personalized driving directions.

In the case of ride-sharing, it's desirable that track trees are pre-built and kept in memory. In order to create and cache a single or multiple track trees with each user's top 5 tracks, a ride-sharing application satisfying 1 million users would require approximately 10 GB of memory. A single server holding all this data could allow a peak load of 35 requests per second, or more servers could be used if higher peak loads need to be handled.

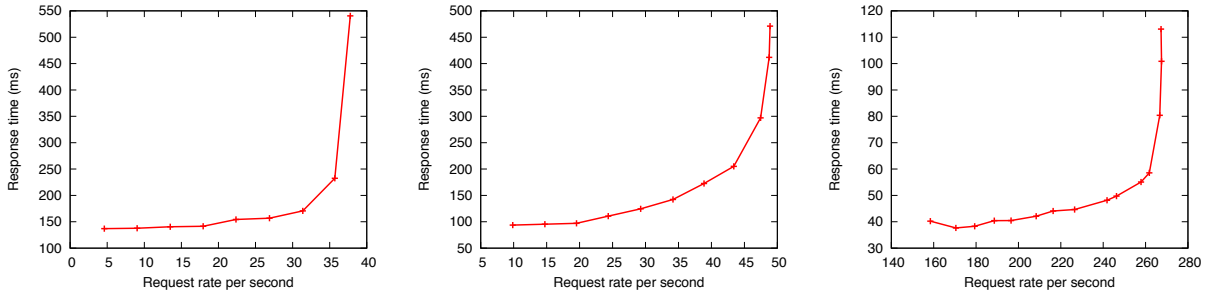


Figure 8: Response times for the RS and PDD applications under varying request rates. (a) RS application; (b) PDD where users' tracks are not cached; (c) PDD where users' tracks are previously cached.

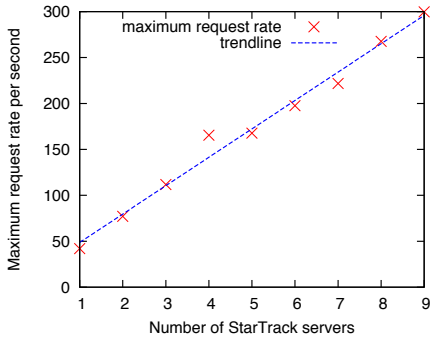


Figure 9: Maximum aggregate request rate with increasing numbers of StarTrack servers.

7 Related Work

As mobile devices have become equipped with the ability to determine their own location, there has been an emergence of applications that collect and utilize users' location data. The research community has proposed a number of useful location-based applications. Traffic prediction [11, 24], ride-sharing [14], personalized driving directions [21] and electronic tour guides [1, 25] are some compelling examples.

At present, every application is forced to maintain its own silo of user location data. StarTrack addresses this problem by providing a common infrastructure that collects location information and enables access to it by multiple applications. In recent years, a number of data platforms (such as Twitter and Facebook) have emerged that enable sharing of information between users. These platforms provide external application developers with an API for accessing user information. StarTrack can be thought of as a platform that stores and enables access to the tracks traversed by users in their daily lives.

Efficient collection of location data is an important precursor to organizing this data and making it accessible. The CarTel project [13] is a distributed sensor network that supports data collection from mobile phones and vehicular sensor networks. CarTel allows applications to visualize traces stored in a relational database using spatial queries.

Database researchers have extensively studied the problem of storing, indexing, and retrieving trajectories. A trajectory is similar to a track in our system and is modeled as a geometric object with 3 dimensions: two for geographical location and a third for time. Prior work has focused on range queries on trajectories and has led to novel indexing techniques. For example, research has shown that it is more efficient to separate the spatial and temporal dimensions and to first index the spatial dimensions [6]. There is also research that optimizes storage and query costs when trajectories are drawn from a fixed road network [2, 4, 22]. Some of the design decisions in StarTrack are based on similar observations. StarTrack additionally allows tracks with very similar geometries to be pruned, resulting in even greater savings. Furthermore, StarTrack exploits the repetitiveness in users' tracks drawn from a road map to implement efficient similarity and common segment queries, which are not studied in previous work.

8 Conclusion

StarTrack enables a broad class of track-based applications, involving both individual users and social networking groups. Our original design of the StarTrack platform focused almost exclusively on the set of operations that would be useful to application developers and ignored performance and scalability considerations. Significant work went into revising the StarTrack design and implementation to enhance its efficiency, robustness, scalability, and ease of use. In some cases, we were able to apply well-known techniques, such as vertical data partitioning and chained declustering. However, most of the observed improvements come from innovative data structures like track trees, new representations for canonicalized tracks,

and novel uses of delayed execution and caching.

The end result is a track-based service that shows several orders of magnitude improvement in performance for operations that are commonly used in the applications that we have developed. This allows such applications to meet their scalability requirements. Moving forward, we plan to build and deploy additional track-based applications to further validate the practical utility of our redesigned service.

Acknowledgments

We thank our former interns Ganesh Ananthanarayanan and Erich Stuntebeck for their work on earlier versions of the system. We thank Lenin Ravindranath and Renato Werneck for helpful discussions; John Krumm, Paul Newson, and Eric Horvitz for providing us with user location data and the map matching software; and Daniel Delling for the shortest path software used to generate synthetic tracks. The anonymous referees and our shepherd, Brad Karp, provided useful suggestions for improving the paper.

References

[1] ABOWD, G. D., ATKESON, C. G., HONG, J., LONG, S., KOOPER, R., AND PINKERTON, M. Cyberguide: A mobile context-aware tour guide. *Wirel. Netw. 3*, 5 (1997), 421–433.

[2] ALMEIDA, V. T. D., AND GÜTING, R. H. Indexing the trajectories of moving objects in networks. *Geoinformatica 9*, 1 (2005), 33–60.

[3] ANANTHANARAYANAN, G., HARIDASAN, M., MOHOMED, I., TERRY, D., AND THEKKATH, C. A. StarTrack: A framework for enabling track-based applications. In *MobiSys '09: Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services* (2009), pp. 207–220.

[4] BRAKATSOULAS, S., PFOSER, D., AND TRYFONA, N. Practical data management techniques for vehicle tracking data. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering* (2005), pp. 324–325.

[5] CAO, L., AND KRUMM, J. From GPS traces to a routable road map. In *GIS '09: Proceedings of 17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems* (2009), pp. 3–12.

[6] CHAKKA, V. P., EVERSPAUGH, A., AND PATEL, J. M. Indexing large trajectory data sets with SETI. In *CIDR '03: 1st Conference on Innovative Data Systems Research* (2003).

[7] CHEN, D., GUIBAS, L. J., HERSHBERGER, J., AND SUN, J. Road network reconstruction for organizing paths. In *SODA '10: Proceedings of 21st ACM-SIAM Symposium on Discrete Algorithms* (2010), pp. 1309–1320.

[8] CHENG, Y.-C., CHAWATHE, Y., LAMARCA, A., AND KRUMM, J. Accuracy characterization for metropolitan-scale wi-fi localization. In *MobiSys '05: Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services* (2005), pp. 233–245.

[9] DAYAL, U., AND BERNSTEIN, P. A. On the updatability of relational views. In *VLDB '78: Proceedings of the 4th International Conference on Very Large Data Bases - Volume 4* (1978), pp. 368–377.

[10] FINKEL, R. A., AND BENTLEY, J. L. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica 4*, 1 (2004), 1–9.

[11] HORVITZ, E., APACIBLE, J., SARIN, R., AND LIAO, L. Prediction, expectation, and surprise: Methods, designs, and study of a deployed traffic forecasting service. In *UAI '05: Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence* (2005), pp. 433–437.

[12] HSIAO, H.-I., AND DEWITT, D. J. Chained declustering: A new availability strategy for multiprocessor database machines. In *ICDE '90: Proceedings of the 6th International Conference on Data Engineering* (1990), pp. 456–465.

[13] HULL, B., BYCHKOVSKY, V., ZHANG, Y., CHEN, K., GORACZKO, M., MIU, A. K., SHIH, E., BALAKRISHNAN, H., AND MADDEN, S. CarTel: A distributed mobile sensor computing system. In *SenSys '06: Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems* (2006), pp. 125–138.

[14] KAMAR, E., AND HORVITZ, E. Collaboration and shared plans in the open world: Studies of ridesharing. In *IJCAI'09: Proceedings of the 21st International Joint Conference on Artifical intelligence* (2009), p. 187.

[15] King county metro transit: Rideshare online http://www.rideshareonline.com, 2010.

[16] KRUMM, J., AND HORVITZ, E. The Microsoft multiperson location survey. Tech. Rep. MSR-TR-2005-103, Microsoft Research, Redmond, WA, USA, 2005.

[17] KRUMM, J., LETCHNER, J., AND HORVITZ, E. Map matching with travel time constraints. In *SAE '07: Proceedings of the Society of Automotive Engineers World Congress* (2007).

[18] LANDIN, P. J. The mechanical evaluation of expressions. *Computer Journal 6* (January 1964), 308–320.

[19] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *ASPLOS '96: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (1996), pp. 84–92.

[20] NEWSON, P., AND KRUMM, J. Hidden Markov map matching through noise and sparseness. In *GIS '09: Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2009), pp. 336–343.

[21] PATEL, K., CHEN, M. Y., SMITH, I., AND LANDAY, J. A. Personalizing routes. In *UIST '06: Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology* (2006), pp. 187–190.

[22] PFOSER, D., AND JENSEN, C. S. Indexing of network constrained moving objects. In *GIS '03: Proceedings of the 11th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems* (2003), pp. 25–32.

[23] SONG, C., QU, Z., BLUMM, N., AND BARABSI, A.-L. Limits of predictability in human mobility. *Science 327*, 5968 (2010), 1018–1021.

[24] THIAGARAJAN, A., RAVINDRANATH, L., LACURTS, K., MADDEN, S., BALAKRISHNAN, H., TOLEDO, S., AND ERIKSSON, J. VTrack: Accurate, energy-aware road traffic delay estimation using mobile phones. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems* (2009), pp. 85–98.

[25] ZHENG, Y., WANG, L., ZHANG, R., XIE, X., AND MA, W.-Y. GeoLife: Managing and understanding your past life over maps. In *MDM '08: Proceedings of the 9th International Conference on Mobile Data Management* (2008), pp. 357–358.

The Turtles Project: Design and Implementation of Nested Virtualization

Muli Ben-Yehuda[†] muli@il.ibm.com
Michael D. Day[‡] mdday@us.ibm.com
Zvi Dubitzky[†] dubi@il.ibm.com
Michael Factor[†] factor@il.ibm.com
Nadav Har'El[†] nyh@il.ibm.com
Abel Gordon[†] abelg@il.ibm.com
Anthony Liguori[‡] aliguori@us.ibm.com
Orit Wasserman[†] oritw@il.ibm.com
Ben-Ami Yassour[†] benami@il.ibm.com
[†]*IBM Research – Haifa*
[‡]*IBM Linux Technology Center*

Abstract

In classical machine virtualization, a hypervisor runs multiple operating systems simultaneously, each on its own virtual machine. In *nested virtualization*, a hypervisor can run multiple other hypervisors with their associated virtual machines. As operating systems gain hypervisor functionality—Microsoft Windows 7 already runs Windows XP in a virtual machine—nested virtualization will become necessary in hypervisors that wish to host them. We present the design, implementation, analysis, and evaluation of high-performance nested virtualization on Intel x86-based systems. The Turtles project, which is part of the Linux/KVM hypervisor, runs multiple *unmodified* hypervisors (e.g., KVM and VMware) and operating systems (e.g., Linux and Windows). Despite the lack of architectural support for nested virtualization in the x86 architecture, it can achieve performance that is within 6-8% of single-level (non-nested) virtualization for common workloads, through *multi-dimensional paging* for MMU virtualization and *multi-level device assignment* for I/O virtualization.

> *The scientist gave a superior smile before replying, "What is the tortoise standing on?" "You're very clever, young man, very clever", said the old lady. "But it's turtles all the way down!"*[1]

1 Introduction

Commodity operating systems increasingly make use of virtualization capabilities in the hardware on which they run. Microsoft's newest operating system, Windows 7, supports a backward compatible Windows XP mode by running the XP operating system as a virtual machine. Linux has built-in hypervisor functionality via the KVM [29] hypervisor. As commodity operating systems gain virtualization functionality, nested virtualization will be required to run those operating systems/hypervisors themselves as virtual machines.

Nested virtualization has many other potential uses. Platforms with hypervisors embedded in firmware [1,20] need to support any workload and specifically other hypervisors as guest virtual machines. An Infrastructure-as-a-Service (IaaS) provider could give a user the ability to run a user-controlled hypervisor as a virtual machine. This way the cloud user could manage his own virtual machines directly with his favorite hypervisor of choice, and the cloud provider could attract users who would like to run their own hypervisors. Nested virtualization could also enable the live migration [14] of hypervisors and their guest virtual machines as a single entity for any reason, such as load balancing or disaster recovery. It also enables new approaches to computer security, such as honeypots capable of running hypervisor-level rootkits [43], hypervisor-level rootkit protection [39,44], and hypervisor-level intrusion detection [18, 25]—for both hypervisors and operating systems. Finally, it could also be used for testing, demonstrating, benchmarking and debugging hypervisors and virtualization setups.

The anticipated inclusion of nested virtualization in x86 operating systems and hypervisors raises many interesting questions, but chief amongst them is its runtime performance cost. Can it be made efficient enough that the overhead doesn't matter? We show that despite the lack of architectural support for nested virtualization in the x86 architecture, efficient nested x86 virtualization—with as little as 6-8% overhead—is feasible even when running *unmodified* binary-only hypervisors executing non-trivial workloads.

Because of the lack of architectural support for nested virtualization, an x86 guest hypervisor cannot use the hardware virtualization support directly to run its own guests. Fundamentally, our approach for nested virtualization *multiplexes* multiple levels of virtualization (mul-

[1] http://en.wikipedia.org/wiki/Turtles_all_the_way_down

tiple hypervisors) on the single level of architectural support available. We address each of the following areas: CPU (e.g., instruction-set) virtualization, memory (MMU) virtualization, and I/O virtualization.

x86 virtualization follows the "trap and emulate" model [21,22,36]. Since every trap by a guest hypervisor or operating system results in a trap to the lowest (most privileged) hypervisor, our approach for CPU virtualization works by having the lowest hypervisor inspect the trap and *forward* it to the hypervisors above it for emulation. We implement a number of optimizations to make world switches between different levels of the virtualization stack more efficient. For efficient memory virtualization, we developed *multi-dimensional paging*, which *collapses* the different memory translation tables into the one or two tables provided by the MMU [13]. For efficient I/O virtualization, we *bypass* multiple levels of hypervisor I/O stacks to provide nested guests with direct assignment of I/O devices [11, 31, 37, 52, 53] via *multi-level device assignment*.

Our main contributions in this work are:

- The design and implementation of nested virtualization for Intel x86-based systems. This implementation can run unmodified hypervisors such as KVM and VMware as guest hypervisors, and can run multiple operating systems such as Linux and Windows as nested virtual machines. Using multi-dimensional paging and multi-level device assignment, it can run common workloads with overhead as low as *6-8%* of single-level virtualization.

- The first evaluation and analysis of nested x86 virtualization performance, identifying the main causes of the virtualization overhead, and classifying them into guest hypervisor issues and limitations in the architectural virtualization support. We also suggest architectural and software-only changes which could reduce the overhead of nested x86 virtualization even further.

2 Related Work

Nested virtualization was first mentioned and theoretically analyzed by Popek and Goldberg [21, 22, 36]. Belpaire and Hsu extended this analysis and created a formal model [10]. Lauer and Wyeth [30] removed the need for a central supervisor and based nested virtualization on the ability to create nested virtual memories. Their implementation required hardware mechanisms and corresponding software support, which bear little resemblance to today's x86 architecture and operating systems.

Belpaire and Hsu also presented an alternative approach for nested virtualization [9]. In contrast to today's x86 architecture which has a single level of architectural support for virtualization, they proposed a hardware architecture with multiple virtualization levels.

The IBM z/VM hypervisor [35] included the first practical implementation of nested virtualization, by making use of multiple levels of architectural support. Nested virtualization was also implemented by Ford et al. in a microkernel setting [16] by modifying the software stack at all levels. Their goal was to enhance OS modularity, flexibility, and extensibility, rather than run unmodified hypervisors and their guests.

During the last decade software virtualization technologies for x86 systems rapidly emerged and were widely adopted by the market, causing both AMD and Intel to add virtualization extensions to their x86 platforms (AMD SVM [4] and Intel VMX [48]). KVM [29] was the first x86 hypervisor to support nested virtualization. Concurrent with this work, Alexander Graf and Joerg Roedel implemented nested support for AMD processors in KVM [23]. Despite the differences between VMX and SVM—VMX takes approximately twice as many lines of code to implement—nested SVM shares many of the same underlying principles as the Turtles project. Multi-dimensional paging was also added to nested SVM based on our work, but multi-level device assignment is not implemented.

There was also a recent effort to incorporate nested virtualization into the Xen hypervisor [24], which again appears to share many of the same underlying principles as our work. It is, however, at an early stage: it can only run a single nested guest on a single CPU, does not have multi-dimensional paging or multi-level device assignment, and no performance results have been published.

Blue Pill [43] is a root-kit based on hardware virtualization extensions. It is loaded during boot time by infecting the disk master boot record. It emulates VMX in order to remain functional and avoid detection when a hypervisor is installed in the system. Blue Pill's nested virtualization support is minimal since it only needs to remain undetectable [17]. In contrast, a hypervisor with nested virtualization support must efficiently multiplex the hardware across multiple levels of virtualization dealing with all of CPU, MMU, and I/O issues. Unfortunately, according to its creators, Blue Pill's nested VMX implementation can not be published.

ScaleMP vSMP is a commercial product which aggregates multiple x86 systems into a single SMP virtual machine. ScaleMP recently announced a new "VM on VM" feature which allows running a hypervisor on top of their underlying hypervisor. No details have been published on the implementation.

Berghmans demonstrates another approach to nested x86 virtualization, where a software-only hypervisor is run on a hardware-assisted hypervisor [12]. In contrast,

our approach allows both hypervisors to take advantage of the virtualization hardware, leading to a more efficient implementation.

3 Turtles: Design and Implementation

The IBM Turtles nested virtualization project implements nested virtualization for Intel's virtualization technology based on the KVM [29] hypervisor. It can host multiple guest hypervisors simultaneously, each with its own multiple nested guest operating systems. We have tested it with unmodified KVM and VMware Server as guest hypervisors, and unmodified Linux and Windows as nested guest virtual machines. Since we treat nested hypervisors and virtual machines as unmodified black boxes, the Turtles project should also run any other x86 hypervisor and operating system.

The Turtles project is fairly mature: it has been tested running multiple hypervisors simultaneously, supports SMP, and takes advantage of two-dimensional page table hardware where available in order to implement nested MMU virtualization via multi-dimensional paging. It also makes use of multi-level device assignment for efficient nested I/O virtualization.

3.1 Theory of Operation

There are two possible models for nested virtualization, which differ in the amount of support provided by the underlying architecture. In the first model, *multi-level architectural support for nested virtualization*, each hypervisor handles all traps caused by sensitive instructions of any guest hypervisor running directly on top of it. This model is implemented for example in the IBM System z architecture [35].

The second model, *single-level architectural support for nested virtualization*, has only a single hypervisor mode, and a trap at any nesting level is handled by this hypervisor. As illustrated in Figure 1, regardless of the level in which a trap occurred, execution returns to the level 0 trap handler. Therefore, any trap occurring at any level from $1 \ldots n$ causes execution to drop to level 0. This limited model is implemented by both Intel and AMD in their respective x86 virtualization extensions, VMX [48] and SVM [4].

Since the Intel x86 architecture is a single-level virtualization architecture, only a single hypervisor can use the processor's VMX instructions to run its guests. For unmodified guest hypervisors to use VMX instructions, this single bare-metal hypervisor, which we call L_0, needs to emulate VMX. This emulation of VMX can work recursively. Given that L_0 provides a faithful emulation of the VMX hardware any time there is a trap on VMX instructions, the guest running on L_1 will not know it is not running directly on the hardware. Building on this infrastructure, the guest at L_1 is itself able use the same techniques to emulate the VMX hardware to an L_2 hypervisor which can then run its L_3 guests. More generally, given that the guest at L_{n-1} provides a faithful emulation of VMX to guests at L_n, a guest at L_n can use the exact same techniques to emulate VMX for a guest at L_{n+1}. We thus limit our discussion below to L_0, L_1, and L_2.

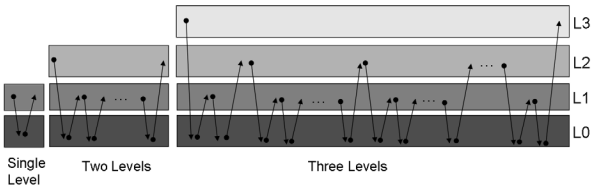


Figure 1: Nested traps with single-level architectural support for virtualization

Fundamentally, our approach for nested virtualization works by *multiplexing* multiple levels of virtualization (multiple hypervisors) on the single level of architectural support for virtualization, as can be seen in Figure 2. Traps are *forwarded* by L_0 between the different levels.

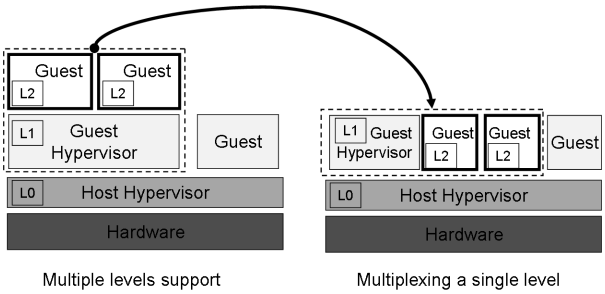


Figure 2: Multiplexing multiple levels of virtualization on a single hardware-provided level of support

When L_1 wishes to run a virtual machine, it launches it via the standard architectural mechanism. This causes a trap, since L_1 is not running in the highest privilege level (as is L_0). To run the virtual machine, L_1 supplies a specification of the virtual machine to be launched, which includes properties such as its initial instruction pointer and its page table root. This specification must be translated by L_0 into a specification that can be used to run L_2 directly on the bare metal, e.g., by converting memory addresses from L_1's physical address space to L_0's physical address space. Thus L_0 *multiplexes* the hardware between L_1 and L_2, both of which end up running as L_0 virtual machines.

When any hypervisor or virtual machine causes a trap, the L_0 trap handler is called. The trap handler then inspects the trapping instruction and its context, and de-

cides whether that trap should be handled by L_0 (e.g., because the trapping context was L_1) or whether to forward it to the responsible hypervisor (e.g., because the trap occurred in L_2 and should be handled by L_1). In the latter case, L_0 forwards the trap to L_1 for handling.

When there are n levels of nesting guests, but the hardware supports less than n levels of MMU or DMA translation tables, the n levels need to be compressed onto the levels available in hardware, as described in Sections 3.3 and 3.4.

3.2 CPU: Nested VMX Virtualization

Virtualizing the x86 platform used to be complex and slow [40, 41, 49]. The hypervisor was forced to resort to on-the-fly binary translation of privileged instructions [3], slow machine emulation [8], or changes to guest operating systems at the source code level [6] or during compilation [32].

In due time Intel and AMD incorporated hardware virtualization extensions in their CPUs. These extensions introduced two new modes of operation: *root mode* and *guest mode*, enabling the CPU to differentiate between running a virtual machine (guest mode) and running the hypervisor (root mode). Both Intel and AMD also added special in-memory virtual machine control structures (VMCS and VMCB, respectively) which contain environment specifications for virtual machines and the hypervisor.

The VMX instruction set and the VMCS layout are explained in detail in [27]. Data stored in the VMCS can be divided into three groups. Guest state holds virtualized CPU registers (e.g., control registers or segment registers) which are automatically loaded by the CPU when switching from root mode to guest mode on VMEntry. Host state is used by the CPU to restore register values when switching back from guest mode to root mode on VMExit. Control data is used by the hypervisor to inject events such as exceptions or interrupts into virtual machines and to specify which events should cause a VMExit; it is also used by the CPU to specify the VMExit reason to the hypervisor.

In nested virtualization, the hypervisor running in root mode (L_0) runs other hypervisors (L_1) in guest mode. L_1 hypervisors have the illusion they are running in root mode. Their virtual machines (L_2) also run in guest mode.

As can be seen in Figure 3, L_0 is responsible for multiplexing the hardware between L_1 and L_2. The CPU runs L_1 using $\text{VMCS}_{0\to1}$ environment specification. Respectively, $\text{VMCS}_{0\to2}$ is used to run L_2. Both of these environment specifications are maintained by L_0. In addition, L_1 creates $\text{VMCS}_{1\to2}$ within its own virtualized environment. Although $\text{VMCS}_{1\to2}$ is never loaded into the processor, L_0 uses it to emulate a VMX enabled CPU for L_1.

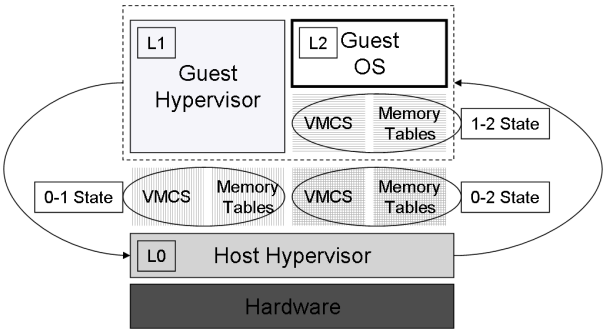


Figure 3: Extending VMX for nested virtualization

3.2.1 VMX Trap and Emulate

VMX instructions can only execute successfully in root mode. In the nested case, L_1 uses VMX instructions in guest mode to load and launch L_2 guests, which causes VMExits. This enables L_0, running in root mode, to trap and emulate the VMX instructions executed by L_1.

In general, when L_0 emulates VMX instructions, it updates VMCS structures according to the update process described in the next section. Then, L_0 resumes L_1, as though the instructions were executed directly by the CPU. Most of the VMX instructions executed by L_1 cause, first, a VMExit from L_1 to L_0, and then a VMEntry from L_0 to L_1.

For the instructions used to run a new VM, `vmresume` and `vmlaunch`, the process is different, since L_0 needs to emulate a VMEntry from L_1 to L_2. Therefore, any execution of these instructions by L_1 cause, first, a VMExit from L_1 to L_0, and then, a VMEntry from L_0 to L_2.

3.2.2 VMCS Shadowing

L_0 prepares a VMCS ($\text{VMCS}_{0\to1}$) to run L_1, exactly in the same way a hypervisor executes a guest with a single level of virtualization. From the hardware's perspective, the processor is running a single hypervisor (L_0) in root mode and a guest (L_1) in guest mode. L_1 is not aware that it is running in guest mode and uses VMX instructions to create the specifications for its own guest, L_2.

L_1 defines L_2's environment by creating a VMCS ($\text{VMCS}_{1\to2}$) which contains L_2's environment from L_1's perspective. For example, the $\text{VMCS}_{1\to2}$ GUEST-CR3 field points to the page tables that L_1 prepared for L_2. L_0 cannot use $\text{VMCS}_{1\to2}$ to execute L_2 directly, since $\text{VMCS}_{1\to2}$ is not valid in L_0's environment and L_0 cannot use L_1's page tables to run L_2. Instead, L_0 uses

$\text{VMCS}_{1\to2}$ to construct a new VMCS ($\text{VMCS}_{0\to2}$) that holds L_2's environment from L_0's perspective.

L_0 must consider all the specifications defined in $\text{VMCS}_{1\to2}$ and also the specifications defined in $\text{VMCS}_{0\to1}$ to create $\text{VMCS}_{0\to2}$. The host state defined in $\text{VMCS}_{0\to2}$ must contain the values required by the CPU to correctly switch back from L_2 to L_0. In addition, $\text{VMCS}_{1\to2}$ host state must be copied to $\text{VMCS}_{0\to1}$ guest state. Thus, when L_0 emulates a switch between L_2 to L_1, the processor loads the correct L_1 specifications.

The guest state stored in $\text{VMCS}_{1\to2}$ does not require any special handling in general, and most fields can be copied directly to the guest state of $\text{VMCS}_{0\to2}$.

The control data of $\text{VMCS}_{1\to2}$ and $\text{VMCS}_{0\to1}$ must be merged to correctly emulate the processor behavior. For example, consider the case where L_1 specifies to trap an event E_A in $\text{VMCS}_{1\to2}$ but L_0 does not trap such event for L_1 (i.e., a trap is not specified in $\text{VMCS}_{0\to1}$). To forward the event E_A to L_1, L_0 needs to specify the corresponding trap in $\text{VMCS}_{0\to2}$. In addition, the field used by L_1 to inject events to L_2 needs to be merged, as well as the fields used by the processor to specify the exit cause.

For the sake of brevity, we omit some details on how specific VMCS fields are merged. For the complete details, the interested reader is encouraged to refer to the KVM source code [29].

3.2.3 VMEntry and VMExit Emulation

In nested environments, switches from L_1 to L_2 and back must be emulated. When L_2 is running and a VMExit occurs there are two possible handling paths, depending on whether the VMExit must be handled only by L_0 or must be forwarded to L_1.

When the event causing the VMExit is related to L_0 only, L_0 handles the event and resumes L_2. This kind of event can be an external interrupt, a non-maskable interrupt (NMI) or any trappable event specified in $\text{VMCS}_{0\to2}$ that was not specified in $\text{VMCS}_{1\to2}$. From L_1's perspective this event does not exist because it was generated outside the scope of L_1's virtualized environment. By analogy to the non-nested scenario, an event occurred at the hardware level, the CPU transparently handled it, and the hypervisor continued running as before.

The second handling path is caused by events related to L_1 (e.g., trappable events specified in $\text{VMCS}_{1\to2}$). In this case L_0 forwards the event to L_1 by copying $\text{VMCS}_{0\to2}$ fields updated by the processor to $\text{VMCS}_{1\to2}$ and resuming L_1. The hypervisor running in L_1 believes there was a VMExit directly from L_2 to L_1. The L_1 hypervisor handles the event and later on resumes L_2 by executing `vmresume` or `vmlaunch`, both of which will be emulated by L_0.

3.3 MMU: Multi-dimensional Paging

In addition to virtualizing the CPU, a hypervisor also needs to virtualize the MMU: A guest OS builds a guest page table which translates guest virtual addresses to guest physical addresses. These must be translated again into host physical addresses. With nested virtualization, a third layer of address translation is needed.

These translations can be done entirely in software, or assisted by hardware. However, as we explain below, current hardware supports only one or two *dimensions* (levels) of translation, not the three needed for nested virtualization. In this section we present a new technique, *multi-dimensional paging*, for multiplexing the three needed translation tables onto the two available in hardware. In Section 4.1.2 we demonstrate the importance of this technique, showing that more naïve approaches (surveyed below) cause at least a three-fold slowdown of some useful workloads.

When no hardware support for memory management virtualization was available, a technique known as shadow page tables [15] was used. A guest creates a guest page table, which translates guest virtual addresses to guest physical addresses. Based on this table, the hypervisor creates a new page table, the *shadow page table*, which translates guest virtual addresses directly to the corresponding host physical address [3, 6]. The hypervisor then runs the guest using this shadow page table instead of the guest's page table. The hypervisor has to trap all guest paging changes, including page fault exceptions, the INVLPG instruction, context switches (which cause the use of a different page table) and all the guest updates to the page table.

To improve virtualization performance, x86 architectures recently added *two-dimensional page tables* [13]—a second translation table in the hardware MMU. When translating a guest virtual address, the processor first uses the regular guest page table to translate it to a guest physical address. It then uses the second table, called EPT by Intel (and NPT by AMD), to translate the guest physical address to a host physical address. When an entry is missing in the EPT table, the processor generates an EPT violation exception. The hypervisor is responsible for maintaining the EPT table and its cache (which can be flushed with INVEPT), and for handling EPT violations, while guest page faults can be handled entirely by the guest.

The hypervisor, depending on the processors capabilities, decides whether to use shadow page tables or two-dimensional page tables to virtualize the MMU. In nested environments, both hypervisors, L_0 and L_1, determine independently the preferred mechanism. Thus, L_0 and L_1 hypervisors can use the same or a different MMU virtualization mechanism. Figure 4 shows three differ-

ent nested MMU virtualization models.

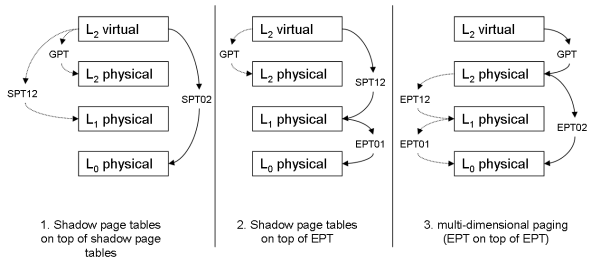


Figure 4: MMU alternatives for nested virtualization

Shadow-on-shadow is used when the processor does not support two-dimensional page tables, and is the least efficient method. Initially, L_0 creates a shadow page table to run L_1 ($SPT_{0\to1}$). L_1, in turn, creates a shadow page table to run L_2 ($SPT_{1\to2}$). L_0 cannot use $SPT_{1\to2}$ to run L_2 because this table translates L_2 guest virtual addresses to L_1 host physical addresses. Therefore, L_0 compresses $SPT_{0\to1}$ and $SPT_{1\to2}$ into a single shadow page table, $SPT_{0\to2}$. This new table translates directly from L_2 guest virtual addresses to L_0 host physical addresses. Specifically, for each guest virtual address in $SPT_{1\to2}$, L_0 creates an entry in $SPT_{0\to2}$ with the corresponding L_0 host physical address.

Shadow-on-EPT is the most straightforward approach to use when the processor supports EPT. L_0 uses the EPT hardware, but L_1 cannot use it, so it resorts to shadow page tables. L_1 uses $SPT_{1\to2}$ to run L_2. L_0 configures the MMU to use $SPT_{1\to2}$ as the first translation table and $EPT_{0\to1}$ as the second translation table. In this way, the processor first translates from L_2 guest virtual address to L_1 host physical address using $SPT_{1\to2}$, and then translates from the L_1 host physical address to the L_0 host physical address using the $EPT_{0\to1}$.

Though the Shadow-on-EPT approach uses the EPT hardware, it still has a noticeable overhead due to page faults and page table modifications in L_2. These must be handled in L_1, to maintain the shadow page table. Each of these faults and writes cause VMExits and must be forwarded from L_0 to L_1 for handling. In other words, Shadow-on-EPT is slow for the exactly the same reasons that Shadow itself was slow for single-level virtualization—but it is even slower because nested exits are slower than non-nested exits.

In *multi-dimensional page tables*, as in two-dimensional page tables, each level creates its own separate translation table. For L_1 to create an EPT table, L_0 exposes EPT capabilities to L_1, even though the hardware only provides a single EPT table.

Since only one EPT table is available in hardware, the two EPT tables should be *compressed* into one: Let us assume that L_0 runs L_1 using $EPT_{0\to1}$, and that L_1 creates an additional table, $EPT_{1\to2}$, to run L_2, because L_0 exposed a virtualized EPT capability to L_1. The L_0 hypervisor could then compress $EPT_{0\to1}$ and $EPT_{1\to2}$ into a single $EPT_{0\to2}$ table as shown in Figure 4. Then L_0 could run L_2 using $EPT_{0\to2}$, which translates directly from the L_2 guest physical address to the L_0 host physical address, reducing the number of page fault exits and improving nested virtualization performance. In Section 4.1.2 we demonstrate more than a three-fold speedup of some useful workloads with multi-dimensional page tables, compared to shadow-on-EPT.

The L_0 hypervisor launches L_2 with an empty $EPT_{0\to2}$ table, building the table on-the-fly, on L_2 EPT-violation exits. These happen when a translation for a guest physical address is missing in the EPT table. If there is no translation in $EPT_{1\to2}$ for the faulting address, L_0 first lets L_1 handle the exit and update $EPT_{1\to2}$. L_0 can now create an entry in $EPT_{0\to2}$ that translates the L_2 guest physical address directly to the L_0 host physical address: $EPT_{1\to2}$ is used to translate the L_2 physical address to a L_1 physical address, and $EPT_{0\to1}$ translates that into the desired L_0 physical address.

To maintain correctness of $EPT_{0\to2}$, the L_0 hypervisor needs to know of any changes that L_1 makes to $EPT_{1\to2}$. L_0 sets the memory area of $EPT_{1\to2}$ as read-only, thereby causing a trap when L_1 tries to update it. L_0 will then update $EPT_{0\to2}$ according to the changed entries in $EPT_{1\to2}$. L_0 also needs to trap all L_1 INVEPT instructions, and invalidate the EPT cache accordingly.

By using *huge pages* [34] to back guest memory, L_0 can create smaller and faster EPT tables. Finally, to further improve performance, L_0 also allows L_1 to use *VPID*s. With this feature, the CPU tags each translation in the TLB with a numeric *virtual-processor id*, eliminating the need for TLB flushes on every VMEntry and VMExit. Since each hypervisor is free to choose these VPIDs arbitrarily, they might collide and therefore L_0 needs to map the VPIDs that L_1 uses into valid L_0 VPIDs.

3.4 I/O: Multi-level Device Assignment

I/O is the third major challenge in server virtualization. There are three approaches commonly used to provide I/O services to a guest virtual machine. Either the hypervisor *emulates* a known device and the guest uses an unmodified driver to interact with it [47], or a *para-virtual driver* is installed in the guest [6, 42], or the host assigns a real device to the guest which then controls the device directly [11, 31, 37, 52, 53]. Device assignment generally provides the best performance [33, 38, 53], since it minimizes the number of I/O-related world switches between the virtual machine and its hypervisor, and although it complicates live migration, device assignment and live

migration can peacefully coexist [26, 28, 54].

These three basic I/O approaches for a single-level guest imply nine possible combinations in the two-level nested guest case. Of the nine potential combinations we evaluated the more interesting cases, presented in Table 1. Implementing the first four alternatives is straightforward. We describe the last option, which we call *multi-level device assignment*, below. Multi-level device assignment lets the L_2 guest access a device directly, bypassing both hypervisors. This direct device access requires dealing with DMA, interrupts, MMIO, and PIOs [53].

I/O virtualization method between L_0 & L_1	I/O virtualization method between L_1 & L_2
Emulation	Emulation
Para-virtual	Emulation
Para-virtual	Para-virtual
Device assignment	Para-virtual
Device assignment	Device assignment

Table 1: I/O combinations for a nested guest

Device DMA in virtualized environments is complicated, because guest drivers use guest physical addresses, while memory access in the device is done with host physical addresses. The common solution to the DMA problem is an IOMMU [2, 11], a hardware component which resides between the device and main memory. It uses a translation table prepared by the hypervisor to translate the guest physical addresses to host physical addresses. IOMMUs currently available, however, only support a single level of address translation. Again, we need to compress two levels of translation tables onto the one level available in hardware.

For modified guests this can be done using a paravirtual IOMMU: the code in L_1 which sets a mapping on the IOMMU from L_2 to L_1 addresses is replaced by a hypercall to L_0. L_0 changes the L_1 address in that mapping to the respective L_0 address, and puts the resulting mapping (from L_2 to L_0 addresses) in the IOMMU.

A better approach, one which can run unmodified guests, is for L_0 to emulate an IOMMU for L_1 [5]. L_1 believes that it is running on a machine with an IOMMU, and sets up mappings from L_2 to L_1 addresses on it. L_0 intercepts these mappings, remaps the L_1 addresses to L_0 addresses, and builds the L_2-to-L_0 map on the real IOMMU.

In current x86 architecture, interrupts always cause a guest exit to L_0, which proceeds to forward the interrupt to L_1. L_1 will then inject it into L_2. The EOI (end of interrupt) will also cause a guest exit. In Section 4.1.1 we discuss the slowdown caused by these interrupt-related exits, and propose ways to avoid it.

Memory-mapped I/O (MMIO) and Port I/O (PIO) for a nested guest work the same way they work for a single-level guest, without incurring exits on the critical I/O path [53].

3.5 Micro Optimizations

There are two main places where a guest of a nested hypervisor is slower than the same guest running on a bare-metal hypervisor. First, the transitions between L_1 and L_2 are slower than the transitions between L_0 and L_1. Second, the exit handling code running in the L_1 hypervisor is slower than the same code running in L_0. In this section we discuss these two issues, and propose optimizations that improve performance. Since we assume that both L_1 and L_2 are unmodified, these optimizations require modifying L_0 only. We evaluate these optimizations in the evaluation section.

3.5.1 Optimizing transitions between L_1 and L_2

As explained in Section 3.2.3, transitions between L_1 and L_2 involve an exit to L_0 and then an entry. In L_0, most of the time is spent merging the VMCS's. We optimize this merging code to only copy data between VMCS's if the relevant values were modified. Keeping track of which values were modified has an intrinsic cost, so one must carefully balance full copying versus partial copying and tracking. We observed empirically that for common workloads and hypervisors, partial copying has a lower overhead.

VMCS merging could be further optimized by copying multiple VMCS fields at once. However, according to Intel's specifications, reads or writes to the VMCS area must be performed using `vmread` and `vmwrite` instructions, which operate on a single field. We empirically noted that under certain conditions one could access VMCS data directly without ill side-effects, bypassing `vmread` and `vmwrite` and copying multiple fields at once with large memory copies. However, this optimization does not strictly adhere to the VMX specifications, and thus might not work on processors other than the ones we have tested.

In the evaluation section, we show that this optimization gives a significant performance boost in micro-benchmarks. However, it did not noticeably improve the other, more typical, workloads that we have evaluated.

3.5.2 Optimizing exit handling in L_1

The exit-handling code in the hypervisor is slower when run in L_1 than the same code running in L_0. The main cause of this slowdown are additional exits caused by privileged instructions in the exit-handling code.

In Intel VMX, the privileged instructions `vmread` and `vmwrite` are used by the hypervisor to read and modify the guest and host specification. As can be seen in Section 4.3, these cause L_1 to exit multiple times while it handles a single L_2 exit.

In contrast, in AMD SVM, guest and host specifications can be read or written to directly using ordinary memory loads and stores. The clear advantage of that model is that L_0 does not intervene while L_1 modifies L_2 specifications. Removing the need to trap and emulate special instructions reduces the number of exits and improves nested virtualization performance.

One thing L_0 can do to avoid trapping on every `vmread` and `vmwrite` is binary translation [3] of problematic `vmread` and `vmwrite` instructions in the L_1 instruction stream, by trapping the first time such an instruction is called and then rewriting it to branch to a non-trapping memory load or store. To evaluate the potential performance benefit of this approach, we tested a modified L_1 that directly reads and writes $\text{VMCS}_{1\to2}$ in memory, instead of using `vmread` and `vmwrite`. The performance of this setup, which we call *DRW* (direct read and write) is described in the evaluation section.

4 Evaluation

We start the evaluation and analysis of nested virtualization with macro benchmarks that represent real-life workloads. Next, we evaluate the contribution of multi-level device assignment and multi-dimensional paging to nested virtualization performance. Most of our experiments are executed with KVM as the L_1 guest hypervisor. In Section 4.2 we present results with VMware Server as the L_1 guest hypervisor.

We then continue the evaluation with a synthetic, worst-case micro benchmark running on L_2 which causes guest exits in a loop. We use this synthetic, worst-case benchmark to understand and analyze the overhead and the handling flow of a single L_2 exit.

Our setup consisted of an IBM x3650 machine booted with a single Intel Xeon 2.9GHz core and with 3GB of memory. The host OS was Ubuntu 9.04 with a kernel that is based on the KVM git tree version kvm-87, with our nested virtualization support added. For both L_1 and L_2 guests we used an Ubuntu Jaunty guest with a kernel that is based on the KVM git tree, version kvm-87. L_1 was configured with 2GB of memory and L_2 was configured with 1GB of memory. For the I/O experiments we used a Broadcom NetXtreme 1Gb/s NIC connected via crossover-cable to an e1000e NIC on another machine.

4.1 Macro Workloads

`kernbench` is a general purpose compilation-type benchmark that compiles the Linux kernel multiple times. The compilation process is, by nature, CPU- and memory-intensive, and it also generates disk I/O to load the compiled files into the guest's page cache.

`SPECjbb` is an industry-standard benchmark designed to measure the server-side performance of Java run-time environments. It emulates a three-tier system and is primarily CPU-intensive.

We executed `kernbench` and `SPECjbb` in four setups: host, single-level guest, nested guest, and nested guest optimized with direct read and write (DRW) as described in Section 3.5.2. The optimizations described in Section 3.5.1 did not make a significant difference to these benchmarks, and are thus omitted from the results. We used KVM as both L_0 and L_1 hypervisor with multi-dimensional paging. The results are depicted in Table 2.

Kernbench				
	Host	Guest	Nested	Nested$_{DRW}$
Run time	324.3	355	406.3	391.5
STD dev.	1.5	10	6.7	3.1
% overhead vs. host	-	9.5	25.3	20.7
% overhead vs. guest	-	-	14.5	10.3
%CPU	93	97	99	99
SPECjbb				
	Host	Guest	Nested	Nested$_{DRW}$
Score	90493	83599	77065	78347
STD dev.	1104	1230	1716	566
% degradation vs. host	-	7.6	14.8	13.4
% degradation vs. guest	-	-	7.8	6.3
%CPU	100	100	100	100

Table 2: `kernbench` and `SPECjbb` results

We compared the impact of running the workloads in a nested guest with running the same workload in a single-level guest, i.e., the overhead added by the additional level of virtualization. For `kernbench`, the overhead of nested virtualization is 14.5%, while for SPECjbb the score is degraded by 7.82%. When we discount the Intel-specific `vmread` and `vmwrite` overhead in L_1, the overhead is 10.3% and 6.3% respectively.

To analyze the sources of overhead, we examine the time distribution between the different levels. Figure 5 shows the time spent in each level. It is interesting to compare the time spent in the hypervisor in the single-level case with the time spent in L_1 in the nested guest

case, since both hypervisors are expected to do the same work. The times are indeed similar, although the L_1 hypervisor takes more cycles due to cache pollution and TLB flushes, as we show in Section 4.3. The significant part of the virtualization overhead in the nested case comes from the time spent in L_0 and the increased number of exits.

For `SPECjbb`, the total number of cycles across all levels is the same for all setups. This is because `SPECjbb` executed for the same pre-set amount of time in both cases and the difference was in the benchmark score.

Efficiently virtualizing a hypervisor is hard. Nested virtualization creates a new kind of workload for the L_0 hypervisor which did not exist before: running another hypervisor (L_1) as a guest. As can be seen in Figure 5, for `kernbench` L_0 takes only 2.28% of the overall cycles in the single-level guest case, but takes 5.17% of the overall cycles for the nested-guest case. In other words, L_0 has to work more than twice as hard when running a nested guest.

Not all exits of L_2 incur the same overhead, as each type of exit requires different handling in L_0 and L_1. In Figure 6, we show the total number of cycles required to handle each exit type. For the single level guest we measured the number of cycles between VMExit and the consequent VMEntry. For the nested guest we measured the number of cycles spent between L_2 VMExit and the consequent L_2 VMEntry.

There is a large variance between the handling times of different types of exits. The cost of each exit comes primarily from the number of privileged instructions performed by L_1, each of which causes an exit to L_0. For example, when L_1 handles a PIO exit of L_2, it generates on average 31 additional exits, whereas in the `cpuid` case discussed later in Section 4.3 only 13 exits are required. Discounting traps due to `vmread` and `vmwrite`, the average number of exits was reduced to 14 for PIO and to 2 for `cpuid`.

Another source of overhead is heavy-weight exits. The external interrupt exit handler takes approximately 64K cycles when executed by L_0. The PIO exit handler takes approximately 12K cycles when executed by L_0. However, when those handlers are executed by L_1, they take much longer: approximately 192K cycles and 183K cycles, respectively. Discounting traps due to `vmread` and `vmwrite`, they take approximately 148K cycles and 130K cycles, respectively. This difference in execution times between L_0 and L_1 is due to two reasons: first, the handlers execute privileged instructions causing exits to L_0. Second, the handlers run for a long time compared with other handlers and therefore more external events such as external interrupts occur during their run-time.

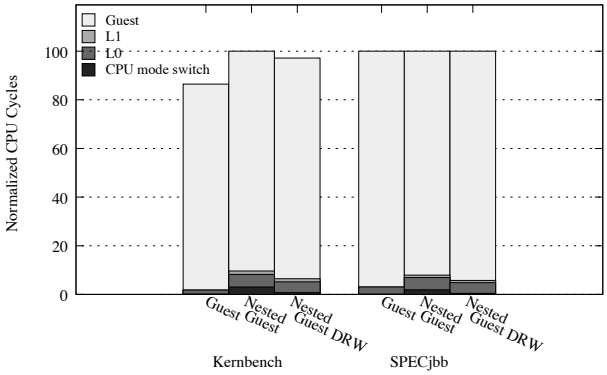


Figure 5: CPU cycle distribution

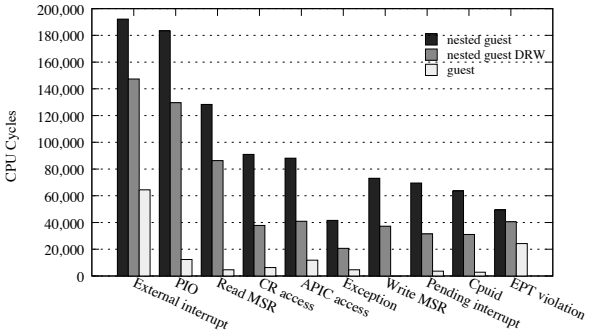


Figure 6: Cycle costs of handling different types of exits

4.1.1 I/O Intensive Workloads

To examine the performance of a nested guest in the case of I/O intensive workloads we used `netperf`, a TCP streaming application that attempts to maximize the amount of data sent over a single TCP connection. We measured the performance on the sender side, with the default settings of `netperf` (16,384 byte messages).

Figure 7 shows the results for running the `netperf` TCP stream test on the host, in a single-level guest, and in a nested guest, using the five I/O virtualization combinations described in Section 3.4. We used KVM's default emulated NIC (RTL-8139), virtio [42] for a paravirtual NIC, and a 1 Gb/s Broadcom NetXtreme II with device assignment. All tests used a single CPU core.

On bare-metal, `netperf` easily achieved line rate (940 Mb/s) with 20% CPU utilization.

Emulation gives a much lower throughput, with full CPU utilization: On a single-level guest we get 25% of the line rate. On the nested guest the throughput is even lower and the overhead is dominated by the cost of device emulation between L_1 and L_2. Each L_2 exit is trapped by L_0 and forwarded to L_1. For each L_2 exit, L_1 then executes multiple privileged instructions, incurring multiple exits back to L_0. In this way the overhead for

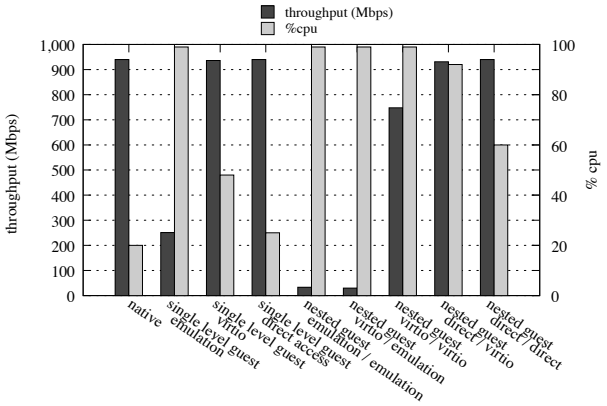


Figure 7: Performance of `netperf` in various setups

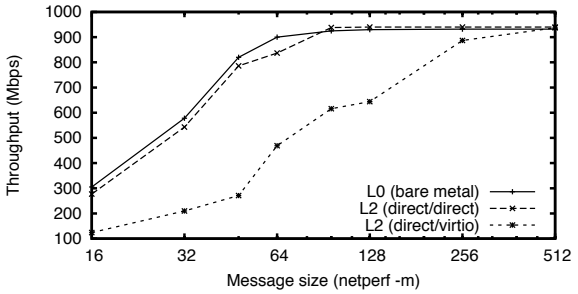


Figure 8: Performance of `netperf` with interrupt-less network driver

each L_2 exit is multiplied.

The para-virtual virtio NIC performs better than emulation since it reduces the number of exits. Using virtio all the way up to L_2 gives 75% of line rate with a saturated CPU, better but still considerably below bare-metal performance.

Multi-level device assignment achieved the best performance, with line rate at 60% CPU utilization (Figure 7, *direct/direct*). Using device assignment between L_0 and L_1 and virtio between L_1 and L_2 enables the L_2 guest to saturate the 1Gb link with 92% CPU utilization (Figure 7, *direct/virtio*).

While multi-level device assignment outperformed the other methods, its measured performance is still suboptimal because 60% of the CPU is used for running a workload that only takes 20% on bare-metal. Unfortunately on current x86 architecture, interrupts cannot be assigned to guests, so both the interrupt itself and its EOI cause exits. The more interrupts the device generates, the more exits, and therefore the higher the virtualization overhead—which is more pronounced in the nested case. We hypothesize that these interrupt-related exits are the biggest source of the remaining overhead, so had the architecture given us a way to avoid these exits—by assigning interrupts directly to guests rather than having each interrupt go through both hypervisors—`netperf` performance on L_2 would be close to that of bare-metal.

To test this hypothesis we reduced the number of interrupts, by modifying standard *bnx2* network driver to work without any interrupts, i.e., continuously poll the device for pending events

Figure 8 compares some of the I/O virtualization combinations with this polling driver. Again, multi-level device assignment is the best option and, as we hypothesized, this time L_2 performance is close to bare-metal. With `netperf`'s default 16,384 byte messages, the throughput is often capped by the 1 Gb/s line rate, so we ran netperf with smaller messages. As we can see in the figure, for 64-byte messages, for example, on L_0 (bare metal) a throughput of 900 Mb/s is achieved, while on L_2 with multi-level device assignment, we get 837 Mb/s, a mere 7% slowdown. The runner-up method, virtio on direct, was not nearly as successful, and achieved just 469 Mb/s, 50% below bare-metal performance. CPU utilization was 100% in all cases since a polling driver consumes all available CPU cycles.

4.1.2 Impact of Multi-dimensional Paging

To evaluate multi-dimensional paging, we compared each of the macro benchmarks described in the previous sections with and without multi-dimensional paging. For each benchmark we configured L_0 to run L_1 with EPT support. We then compared the case where L_1 uses shadow page tables to run L_2 ("Shadow-on-EPT") with the case of L_1 using EPT to run L_2 ("multi-dimensional paging").

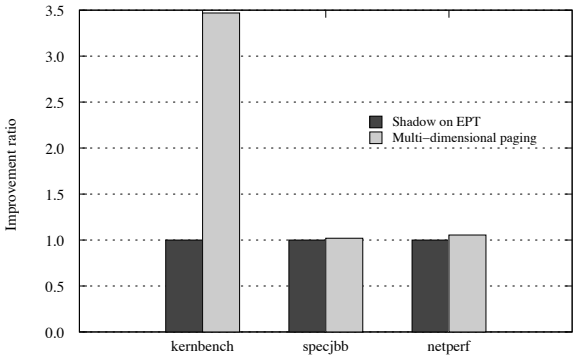


Figure 9: Impact of multi-dimensional paging

Figure 9 shows the results. The overhead between the two cases is mostly due to the number of page-fault exits. When shadow paging is used, each page fault of the L_2 guest results in a VMExit. When multi-dimensional pag-

ing is used, only an access to a guest physical page that is not mapped in the EPT table will cause an EPT violation exit. Therefore the impact of multi-dimensional paging depends on the number of guest page faults, which is a property of the workload. The improvement is startling in benchmarks such as `kernbench` with a high number of page faults, and is less pronounced in workloads that do not incur many page faults.

4.2 VMware Server as a Guest Hypervisor

We also evaluated VMware as the L_1 hypervisor to analyze how a different guest hypervisor affects nested virtualization performance. We used the hosted version, VMWare Server v2.0.1, build 156745 x86-64, on top of Ubuntu based on kernel 2.6.28-11. We intentionally did not install VMware tools for the L_2 guest, thereby increasing nested virtualization overhead. Due to similar results obtained for VMware and KVM as the nested hypervisor, we show only `kernbench` and `SPECjbb` results below.

Benchmark	% overhead vs. single-level guest
`kernbench`	14.98
`SPECjbb`	8.85

Table 3: VMware Server as a guest hypervisor

Examining L_1 exits, we noticed VMware Server uses VMX initialization instructions (`vmon`, `vmoff`, `vmptrld`, `vmclear`) several times during L_2 execution. Conversely, KVM uses them only once. This dissimilitude derives mainly from the approach used by VMware to interact with the host Linux kernel. Each time the monitor module takes control of the CPU, it enables VMX. Then, before it releases control to the Linux kernel, VMX is disabled. Furthermore, during this transition many non-VMX privileged instructions are executed by L_1, increasing L_0 intervention.

Although all these initialization instructions are emulated by L_0, transitions from the VMware monitor module to the Linux kernel are less frequent for Kernbench and SPECjbb. The VMware monitor module typically handles multiple L_2 exits before switching to the Linux kernel. As a result, this behavior only slightly affected the nested virtualization performance.

4.3 Micro Benchmark Analysis

To analyze the cycle-costs of handling a single L_2 exit, we ran a micro benchmark in L_2 that does nothing except generate exits by calling `cpuid` in a loop. The virtualization overhead for running an L_2 guest is the ratio between the effective work done by the L_2 guest and the overhead of handling guest exits in L_0 and L_1. Based on this definition, this `cpuid` micro benchmark is a worst case workload, since L_2 does virtually nothing except generate exits. We note that `cpuid` cannot in the general case be handled by L_0 directly, as L_1 may wish to modify the values returned to L_2.

Figure 10 shows the number of CPU cycles required to execute a single `cpuid` instruction. We ran the `cpuid` instruction $4 * 10^6$ times and calculated the average number of cycles per iteration. We repeated the test for the following setups: 1. native, 2. running `cpuid` in a single level guest, and 3. running `cpuid` in a nested guest with and without the optimizations described in Section 3.5. For each execution, we present the distribution of the cycles between the levels: L_0, L_1, L_2. CPU mode switch stands for the number of cycles spent by the CPU when performing a VMEntry or a VMExit. On bare metal `cpuid` takes about 100 cycles, while in a virtual machine it takes about 2,600 cycles (Figure 10, *column 1*), about 1,000 of which is due to the CPU mode switching. When run in a nested virtual machine it takes about 58,000 cycles (Figure 10, *column 2*).

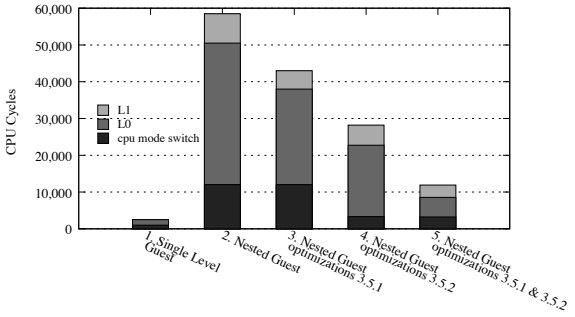


Figure 10: CPU cycle distribution for `cpuid`

To understand the cost of handling a nested guest exit compared to the cost of handling the same exit for a single-level guest, we analyzed the flow of handling `cpuid`:

1. L_2 executes a `cpuid` instruction
2. CPU traps and switches to root mode L_0
3. L_0 switches state from running L_2 to running L_1
4. CPU switches to guest mode L_1
5. L_1 modifies $\text{VMCS}_{1\to2}$
   repeat n times:
   (a) L_1 accesses $\text{VMCS}_{1\to2}$
   (b) CPU traps and switches to root mode L_0
   (c) L_0 emulates $\text{VMCS}_{1\to2}$ access and resumes L_1
   (d) CPU switches to guest mode L_1
6. L_1 emulates `cpuid` for L_2

7. L_1 executes a resume of L_2
8. CPU traps and switches to root mode L_0
9. L_0 switches state from running L_1 to running L_2
10. CPU switches to guest mode L_2

In general, step 5 can be repeated multiple times. Each iteration consists of a single VMExit from L_1 to L_0. The total number of exits depends on the specific implementation of the L_1 hypervisor. A nesting-friendly hypervisor will keep privileged instructions to a minimum. In any case, the L_1 hypervisor must interact with $\text{VMCS}_{1\to2}$, as described in Section 3.2.2. In the case of `cpuid`, in step 5, L_1 reads 7 fields of $\text{VMCS}_{1\to2}$, and writes 4 fields to $\text{VMCS}_{1\to2}$, which ends up as 11 VMExits from L_1 to L_0. Overall, for a single L_2 `cpuid` exit there are 13 CPU mode switches from guest mode to root mode and 13 CPU mode switches from root mode to guest mode, specifically in steps: 2, 4, 5b, 5d, 8, 10.

The number of cycles the CPU spends in a single switch to guest mode plus the number of cycles to switch back to root mode, is approximately 1,000. The total CPU switching cost is therefore around 13,000 cycles.

The other two expensive steps are 3 and 9. As described in Section 3.5, these switches can be optimized. Indeed as we show in Figure 10, *column 3*, using various optimizations we can reduce the virtualization overhead by 25%, and by 80% when using non-trapping `vmread` and `vmwrite` instructions.

By avoiding traps on `vmread` and `vmwrite` (*Figure 10, columns 4 and 5*), we removed the exits caused by $\text{VMCS}_{1\to2}$ accesses and the corresponding VMCS access emulation, step 5. This optimization reduced the switching cost by 84.6%, from 13,000 to 2,000.

While it might still be possible to optimize steps 3 and 9 further, it is clear that the exits of L_1 while handling a single exit of L_2, and specifically VMCS accesses, are a major source of overhead. Architectural support for both faster world switches and VMCS updates without exits will reduce the overhead.

Examining Figure 10, it seems that handling `cpuid` in L_1 is more expensive than handling `cpuid` in L_0. Specifically, in column 3, the nested hypervisor L_1 spends around 5,000 cycles to handle `cpuid`, while in column 1 the same hypervisor running on bare metal only spends 1500 cycles to handle the same exit (note that these numbers do not include the mode switches). The code running in L_1 and in L_0 is identical; the difference in cycle count is due to cache pollution. Running the `cpuid` handling code incurs on average 5 L2 cache misses and 2 TLB misses when run in L_0, whereas running the exact same code in L_1 incurs on average 400 L2 cache misses and 19 TLB misses.

5 Discussion

In nested environments we introduce a new type of workload not found in single-level virtualization: the hypervisor as a guest. Traditionally, x86 hypervisors were designed and implemented assuming they will be running directly on bare metal. When they are executed on top of another hypervisor this assumption no longer holds and the guest hypervisor behavior becomes a key factor.

With a nested L_1 hypervisor, the cost of a single L_2 exit depends on the number of exits caused by L_1 during the L_2 exit handling. A nesting-friendly L_1 hypervisor should minimize this critical chain to achieve better performance, for example by limiting the use of trap-causing instructions in the critical path.

Another alternative for reducing this critical chain is to para-virtualize the guest hypervisor, similar to OS paravirtualization [6, 50, 51]. While this approach could reduce L_0 intervention when L_1 virtualizes the L_2 environment, the work being done by L_0 to virtualize the L_1 environment will still persist. How much this technique can help depends on the workload and on the specific approach used. Taking as a concrete example the conversion of `vmread`s and `vmwrite`s to non-trapping load/stores, para-virtualization could reduce the overhead for `kernbench` from 14.5% to 10.3%.

5.1 Architectural Overhead

Part of the overhead introduced with nested virtualization is due to the architectural design choices of x86 hardware virtualization extensions.

Virtualization API: Two performance sensitive areas in x86 virtualization are memory management and I/O virtualization. With multi-dimensional paging we compressed three MMU translation tables onto the two available in hardware; multi-level device assignment does the same for IOMMU translation tables. Architectural support for multiple levels of MMU and DMA translation tables—as many tables as there are levels of nested hypervisors—will immediately improve MMU and I/O virtualization.

Architectural support for delivering interrupts directly from the hardware to the L_2 guest will remove L_0 intervention on interrupt delivery and completion, intervention which, as we explained in Section 4.1.1, hurts nested performance. Such architectural support will also help single-level I/O virtualization performance [33].

VMX features such as MSR bitmaps, I/O bitmaps, and CR masks/shadows [48] proved to be effective in reducing exit overhead. Any architectural feature that reduces single-level exit overhead also shortens the nested critical path. Such features, however, also add implementation complexity, since to exploit them in nested environments

they must be properly emulated by L_0 hypervisors.

Removing the (Intel-specific) need to trap on every `vmread` and `vmwrite` instruction will give an immediate performance boost, as we showed in Section 3.5.2.

Same Core Constraint: The x86 trap-and-emulate implementation dictates that the guest and hypervisor share each core, since traps are always handled on the core where they occurred. Due to this constraint, when the hypervisor handles an exit the guest is temporarily stopped on that core. In a nested environment, the L_1 guest hypervisor will also be interrupted, increasing the total interruption time of the L_2 guest. Gavrilovska, et al., presented techniques for exploiting additional cores to handle guest exits [19]. According to the authors, for a single level of virtualization, they measured 41% average improvements in call latency for null calls, cpuid and page table updates. These techniques could be adapted for nested environments in order to remove L_0 interventions and also reduce privileged instructions call latencies, decreasing the total interruption time of a nested guest.

Cache Pollution: Each time the processor switches between the guest and the host context on a single core, the effectiveness of its caches is reduced. This phenomenon is magnified in nested environments, due to the increased number of switches. As was seen in Section 4.3, even after discounting L_0 intervention, the L_1 hypervisor still took more cycles to handle an L_2 exit than it took to handle the same exit for the single-level scenario, due to cache pollution. Dedicating cores to guests could reduce cache pollution [7, 45, 46] and increase performance.

6 Conclusions and Future Work

Efficient nested x86 virtualization is feasible, despite the challenges stemming from the lack of architectural support for nested virtualization. Enabling efficient nested virtualization on the x86 platform through multi-dimensional paging and multi-level device assignment opens exciting avenues for exploration in such diverse areas as security, clouds, and architectural research.

We are continuing to investigate architectural and software-based methods to improve the performance of nested virtualization, while simultaneously exploring ways of building computer systems that have nested virtualization built-in.

Last, but not least, while the Turtles project is fairly mature, we expect that the additional public exposure stemming from its open source release will help enhance its stability and functionality. We look forward to seeing in what interesting directions the research and open source communities will take it.

Acknowledgments

The authors would like to thank Alexander Graf and Joerg Roedel, whose KVM patches for nested SVM inspired parts of this work. The authors would also like to thank Ryan Harper, Nadav Amit, and our shepherd Robert English for insightful comments and discussions.

References

[1] Phoenix Hyperspace. http://www.hyperspace.com/.

[2] ABRAMSON, D., JACKSON, J., MUTHRASANALLUR, S., NEIGER, G., REGNIER, G., SANKARAN, R., SCHOINAS, I., UHLIG, R., VEMBU, B., AND WIEGERT, J. Intel virtualization technology for directed I/O. *Intel Technology Journal 10*, 03 (August 2006), 179–192.

[3] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. *SIGOPS Oper. Syst. Rev. 40*, 5 (December 2006), 2–13.

[4] AMD. Secure virtual machine architecture reference manual.

[5] AMIT, N., BEN-YEHUDA, M., AND YASSOUR, B.-A. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *WIOSCA '10: Sixth Annual Workshop on the Interaction between Operating Systems and Computer Architecture*.

[6] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP '03: Symposium on Operating Systems Principles* (2003).

[7] BAUMANN, A., BARHAM, P., DAGAND, P. E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new os architecture for scalable multicore systems. In *SOSP '09: 22nd ACM SIGOPS Symposium on Operating systems principles*, pp. 29–44.

[8] BELLARD, F. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference* (2005), p. 41.

[9] BELPAIRE, G., AND HSU, N.-T. Hardware architecture for recursive virtual machines. In *ACM '75: 1975 annual ACM conference*, pp. 14–18.

[10] BELPAIRE, G., AND HSU, N.-T. Formal properties of recursive virtual machine architectures. *SIGOPS Oper. Syst. Rev. 9*, 5 (1975), 89–96.

[11] BEN-YEHUDA, M., MASON, J., XENIDIS, J., KRIEGER, O., VAN DOORN, L., NAKAJIMA, J., MALLICK, A., AND WAHLIG, E. Utilizing IOMMUs for virtualization in Linux and Xen. In *OLS '06: The 2006 Ottawa Linux Symposium*, pp. 71–86.

[12] BERGHMANS, O. Nesting virtual machines in virtualization test frameworks. Master's thesis, University of Antwerp, May 2010.

[13] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *ASPLOS '08: 13th intl. conference on architectural support for programming languages and operating systems* (2008).

[14] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *NSDI '05: Second Symposium on Networked Systems Design & Implementation* (2005), pp. 273–286.

[15] DEVINE, S. W., BUGNION, E., AND ROSENBLUM, M. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. US #6397242, May 2002.

[16] FORD, B., HIBLER, M., LEPREAU, J., TULLMANN, P., BACK, G., AND CLAWSON, S. Microkernels meet recursive virtual machines. In *OSDI '96: Second USENIX symposium on Operating systems design and implementation* (1996), pp. 137–151.

[17] GARFINKEL, T., ADAMS, K., WARFIELD, A., AND FRANKLIN, J. Compatibility is not transparency: VMM detection myths and realities. In *HOTOS'07: 11th USENIX workshop on Hot topics in operating systems* (2007), pp. 1–6.

[18] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *Network & Distributed Systems Security Symposium* (2003), pp. 191–206.

[19] GAVRILOVSKA, A., KUMNAR, S., RAJ, H., SCHWAN, K., GUPTA, V., NATHUJI, R., NIRANJAN, R., RANADIVE, A., AND SARAIYA, P. High-performance hypervisor architectures: Virtualization in hpc systems. In *HPCVIRT '07: 1st Workshop on System-level Virtualization for High Performance Computing*.

[20] GEBHARDT, C., AND DALTON, C. Lala: a late launch application. In *STC '09: 2009 ACM workshop on Scalable trusted computing* (2009), pp. 1–8.

[21] GOLDBERG, R. P. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems* (New York, NY, USA, 1973), ACM, pp. 74–112.

[22] GOLDBERG, R. P. Survey of virtual machine research. *IEEE Computer Magazine* (June 1974), 34–45.

[23] GRAF, A., AND ROEDEL, J. Nesting the virtualized world. Linux Plumbers Conference, Sep. 2009.

[24] HE, Q. Nested virtualization on xen. Xen Summit Asia 2009.

[25] HUANG, J.-C., MONCHIERO, M., AND TURNER, Y. Ally: Os-transparent packet inspection using sequestered cores. In *WIOV '10: The Second Workshop on I/O Virtualization*.

[26] HUANG, W., LIU, J., KOOP, M., ABALI, B., AND PANDA, D. Nomad: migrating OS-bypass networks in virtual machines. In *VEE '07: 3rd international conference on Virtual execution environments* (2007), pp. 158–168.

[27] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developers Manual*. 2009.

[28] KADAV, A., AND SWIFT, M. M. Live migration of direct-access devices. In *First Workshop on I/O Virtualization (WIOV '08)*.

[29] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the linux virtual machine monitor. In *Ottawa Linux Symposium* (July 2007), pp. 225–230.

[30] LAUER, H. C., AND WYETH, D. A recursive virtual machine architecture. In *Workshop on virtual computer systems* (1973), pp. 113–116.

[31] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GÖTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI '04: 6th conference on Symposium on Opearting Systems Design & Implementation* (2004), p. 2.

[32] LEVASSEUR, J., UHLIG, V., YANG, Y., CHAPMAN, M., CHUBB, P., LESLIE, B., AND HEISER, G. Pre-virtualization: Soft layering for virtual machines. In *ACSAC '08: 13th Asia-Pacific Computer Systems Architecture Conference*, pp. 1–9.

[33] LIU, J. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *IPDPS '10: IEEE International Parallel and Distributed Processing Symposium* (2010).

[34] NAVARRO, J., IYER, S., DRUSCHEL, P., AND COX, A. Practical, transparent operating system support for superpages. In *OSDI '02: 5th symposium on Operating systems design and implementation* (2002), pp. 89–104.

[35] OSISEK, D. L., JACKSON, K. M., AND GUM, P. H. Esa/390 interpretive-execution architecture, foundation for vm/esa. *IBM Systems Journal 30*, 1 (1991).

[36] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17*, 7 (July 1974), 412–421.

[37] RAJ, H., AND SCHWAN, K. High performance and scalable I/O virtualization via self-virtualized devices. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing* (2007), pp. 179–188.

[38] RAM, K. K., SANTOS, J. R., TURNER, Y., COX, A. L., AND RIXNER, S. Achieving 10Gbps using safe and transparent network interface virtualization. In *VEE '09: The 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (March 2009).

[39] RILEY, R., JIANG, X., AND XU, D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Recent Advances in Intrusion Detection*, vol. 5230 of *Lecture Notes in Computer Science*. 2008, ch. 1, pp. 1–20.

[40] ROBIN, J. S., AND IRVINE, C. E. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *9th conference on USENIX Security Symposium* (2000), p. 10.

[41] ROSENBLUM, M. Vmware's virtual platform: A virtual machine monitor for commodity pcs. In *Hot Chips 11* (1999).

[42] RUSSELL, R. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev. 42*, 5 (2008), 95–103.

[43] RUTKOWSKA, J. Subverting vista kernel for fun and profit. Blackhat, Aug. 2006.

[44] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: 21st ACM SIGOPS symposium on Operating systems principles* (2007), pp. 335–350.

[45] SHALEV, L., BOROVIK, E., SATRAN, J., AND BEN-YEHUDA, M. Isostack—highly efficient network processing on dedicated cores. In *USENIX ATC '10: The 2010 USENIX Annual Technical Conference* (2010).

[46] SHALEV, L., MAKHERVAKS, V., MACHULSKY, Z., BIRAN, G., SATRAN, J., BEN-YEHUDA, M., AND SHIMONY, I. Loosely coupled tcp acceleration architecture. In *HOTI '06: Proceedings of the 14th IEEE Symposium on High-Performance Interconnects* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 3–8.

[47] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference* (2001).

[48] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A. L., MARTINS, F. C. M., ANDERSON, A. V., BENNETT, S. M., KAGI, A., LEUNG, F. H., AND SMITH, L. Intel virtualization technology. *Computer 38*, 5 (2005), 48–56.

[49] WALDSPURGER, C. A. Memory resource management in VMware ESX server. In *OSDI '02: 5th Symposium on Operating System Design and Implementation*.

[50] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Denali: a scalable isolation kernel. In *EW '10: 10th ACM SIGOPS European workshop* (2002), pp. 10–15.

[51] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 195–209.

[52] WILLMANN, P., SHAFER, J., CARR, D., MENON, A., RIXNER, S., COX, A. L., AND ZWAENEPOEL, W. Concurrent direct network access for virtual machine monitors. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on* (2007), pp. 306–317.

[53] YASSOUR, B.-A., BEN-YEHUDA, M., AND WASSERMAN, O. Direct device assignment for untrusted fully-virtualized virtual machines. Tech. rep., IBM Research Report H-0263, 2008.

[54] ZHAI, E., CUMMINGS, G. D., AND DONG, Y. Live migration with pass-through device for Linux VM. In *OLS '08: The 2008 Ottawa Linux Symposium* (July 2008), pp. 261–268.

mClock: Handling Throughput Variability for Hypervisor IO Scheduling

Ajay Gulati
VMware Inc.
Palo Alto, CA, 94304
agulati@vmware.com

Arif Merchant
HP Labs
Palo Alto, CA 94304
arif.merchant@acm.org

Peter J. Varman
Rice University
Houston, TX, 77005
pjv@rice.edu

Abstract

Virtualized servers run a diverse set of virtual machines (VMs), ranging from interactive desktops to test and development environments and even batch workloads. Hypervisors are responsible for multiplexing the underlying hardware resources among VMs while providing them the desired degree of isolation using resource management controls. Existing methods provide many knobs for allocating CPU and memory to VMs, but support for control of IO resource allocation has been quite limited. IO resource management in a hypervisor introduces significant new challenges and needs more extensive controls than in commodity operating systems.

This paper introduces a novel algorithm for IO resource allocation in a hypervisor. Our algorithm, *mClock*, supports proportional-share fairness subject to minimum reservations and maximum limits on the IO allocations for VMs. We present the design of mClock and a prototype implementation inside the VMware ESX server hypervisor. Our results indicate that these rich QoS controls are quite effective in isolating VM performance and providing better application latency. We also show an adaptation of mClock (called *dmClock*) for a distributed storage environment, where storage is jointly provided by multiple nodes.

1 Introduction

The increasing trend towards server virtualization has elevated hypervisors to first class entities in today's datacenters. Virtualized hosts run tens to hundreds of virtual machines (VMs), and the hypervisor needs to provide each virtual machine with the illusion of owning dedicated physical resources: CPU, memory, network and storage IO. Strong isolation is needed for successful consolidation of VMs with diverse requirements on a shared infrastructure. Existing products such as VMware ESX server hypervisor provide guarantees for CPU and memory allocation using sophisticated controls such as reservations, limits and shares [3, 44]. However, the current state of the art in storage IO resource allocation is much more rudimentary, limited to providing proportional shares [20] to different VMs.

IO scheduling in a hypervisor introduces many new challenges compared to managing other shared resources. First, virtualized servers typically access a shared storage device using either a clustered file system such as VMFS [11] or NFS volumes. A storage device in the guest OS or a VM is just a large file on the shared storage device. Second, the IO scheduler in the hypervisor runs one layer below the elevator-based scheduling in the guest OS. Hence, it needs to handle issues such as locality of accesses across VMs, high variability in IO sizes, different request priorities based on the applications running in the VMs, and bursty workloads.

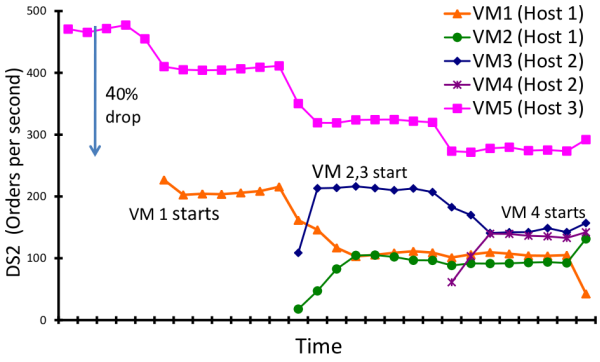


Figure 1: Orders/sec for VM5 decreases as the load on the shared storage device increases from VMs running on other hosts.

In addition, the amount of IO throughput available to any particular host can fluctuate widely based on the behavior of other hosts accessing the shared device. Unlike CPU and memory resources, the IO throughput available to a host is not under its own control. As shown in the example below, this can cause large variations in the IOPS available to a VM and impact application-level performance.

Consider the simple scenario shown in Figure 1, with three hosts and five VMs. Each VM is running a DVD-Store [2] benchmark, which is an IO-intensive OLTP workload. The system administrator has carefully provisioned the resources (CPU and memory) needed by VM 5, so that it can serve at least 400 orders per second. Initially, VM 5 is running on host 3, and it achieves a transaction rate of roughly 500 orders/second. Later, as four other VMs (1 – 4), running on two separate hosts sharing the same storage device, start to consume IO

bandwidth, the transaction rate of VM 5 drops to 275 orders per second, which is significantly lower than expected. Other events that can cause this sort of fluctuation are: (1) changes in workloads (2) background tasks scheduled at the storage array, and (3) changes in SAN paths between the hosts and storage device.

PARDA [20] provided a distributed control algorithm to allocate queue slots at the storage device to hosts in proportion to the aggregate IO shares of the VMs running on them. The local IO scheduling at each host was done using SFQ(D) [24] a traditional fair-scheduler, which divides the aggregate host throughput among the VMs in proportion to their shares. Unfortunately, as aggregate throughput fluctuates downwards, or as the value of a VM's shares is diluted by the addition of other VMs to the system, the absolute throughput for a VM falls. This open-ended dilution is unacceptable in many applications that require minimum resource requirements to function. Lack of QoS support for IO resources can have widespread effects, rendering existing CPU and memory controls ineffective when applications block on IO requests. Arguably, this limitation is one of the reasons for the slow adoption of IO-intensive applications in virtualized environments.

Resource controls such as *shares* (a.k.a. weights), *reservations*, and *limits* are used for predictable service allocation with strong isolation [8, 34, 43, 44]. Shares are a relative allocation measure that specify the ratio in which the different VMs receive service. Reservations and limits are expressed in absolute units, *e.g.* CPU cycles/sec or megabytes of memory. The general idea is to allocate the resource to the VMs in proportion to their shares, subject to the constraints that each VM receives at least its reservation and no more than its limit. These controls have primarily been employed for allocating resources like CPU time and memory pages where the resource capacity is known and fixed.

For fixed-capacity resources, one can combine *shares* and *reservations* into one single allocation for a VM. This allocation can be calculated whenever a new VM enters or leaves the system, since these are the only events at which the allocation is affected. However, enforcing these controls is much more difficult when the capacity fluctuates dynamically, as is the case for the IO bandwidth of shared storage. In this case the allocations need to be continuously monitored (rather than only at VM entry and exit) to ensure that no VM falls below its minimum. A brute-force solution is to emulate the method used for fixed-capacity resources by recomputing the allocations periodically. However this method relies on accurately being able to predict future capacity based on the current state.

Finally, *limits* provide an upper bound on the absolute resource allocations. Such a limit on IO performance

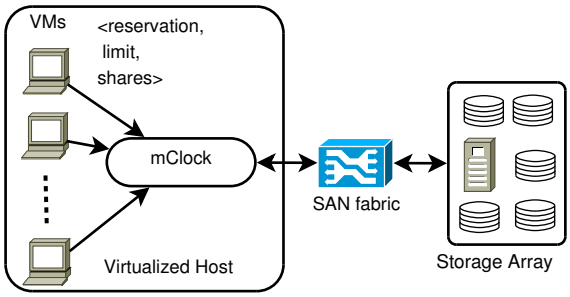


Figure 2: Virtualized host with VMs accessing a shared storage array over a SAN

is desirable to prevent competing IO-intensive applications, such as virus scanners, virtual-disk migrations, or backup operations, from consuming all the spare bandwidth in the system, which can result in high latencies for bursty and ON-OFF workloads. There are yet other reasons cited by service providers for wanting to explicitly limit IO throughput; for example, to avoid giving VMs more throughput than has been paid for, or to avoid raising expectations on performance that cannot generally be sustained [1, 8].

In this paper, we present *mClock*, an IO scheduler that provides all three controls mentioned above at a per-VM level (Figure 2). We believe that mClock is the first scheduler to provide such controls in the presence of capacity fluctuations at short time scales. We have implemented *mClock*, along with certain storage-specific optimizations, as a prototype scheduler in the VMware ESX server hypervisor and showed its effectiveness for various use cases.

We also demonstrate *dmClock,* a distributed version of the algorithm that can be used in clustered storage systems, where the storage is distributed across multiple nodes (*e.g.*, LeftHand [4], Seanodes [6], IceCube [46], FAB [30]). dmClock ensures that the overall allocation to each VM is based on the specified shares, reservations, and limits even when the VM load is non-uniformly distributed across the storage nodes.

The remainder of the paper is organized as follows. In Section 2 we discuss mClock's scheduling goal and its comparison with existing approaches. Section 3 presents the mClock algorithm in detail, along with storage-specific optimizations. Distributed implementation for a clustered storage system is discussed in Section 3.2. Detailed performance evaluation using a diverse set of workloads is presented in Section 4. Finally we conclude with some directions for future work in Section 5.

2 Overview and Related Work

The work related to QoS-based IO resource allocation can be divided into three broad areas. First is the class of algorithms that provide proportional allocation of IO

Algorithm class	Proportional allocation	Latency support	Reservation Support	Limit Support	Handle Capacity fluctuation
Proportional Sharing (PS) Algorithms	Yes	No	No	No	No
PS + Latency support	Yes	Yes	No	No	No
PS + Reservations	Yes	Yes	Yes	No	No
mClock	Yes	Yes	Yes	Yes	Yes

Table 1: Comparison of mClock with existing scheduling techniques

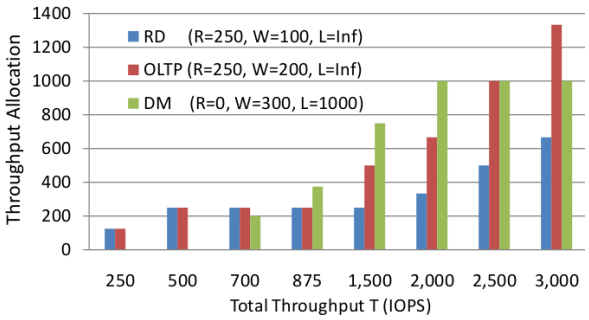


Figure 3: Allocation of IOPS to various VMs as the overall throughput changes

resources, such as Stonehenge [23] SFQ(D) [24], Argon [41], and Aqua [48]. Many of these algorithms are variants of weighted fair queuing mechanisms (Virtual Clock [50], WFQ [13], PGPS [29], WF^2Q [10], SCFQ [15], Leap Forward [38], SFQ [18] and Latency-rate scheduling [33]) proposed in the networking literature, adapted to handle various storage-specific concerns such as concurrency, minimizing seek delays and improving throughput.

The goal of these algorithms is to allocate throughput or bandwidth in proportion to the specified weights of the clients. Second is the class of algorithms that provide support for latency-sensitive applications along with proportional sharing. These algorithms include SMART [28], BVT [14], pClock [22], Avatar [49] and service curve based techniques [12, 27, 31, 36]. Third is the class of algorithms that support reservation along with proportional allocation, such as Rialto [25], ESX memory management [44] and other reservation based CPU scheduling methods [17, 34, 35]. Table 1 provides a quick comparison of mClock with existing algorithms in the three categories.

2.1 Scheduling Goals of mClock

We first discuss a simple example describing the scheduling policy of mClock. As mentioned earlier, three parameters are specified for each VM in the system: a *share* or *weight* represented by w_i, a *reservation* r_i, and a *limit* l_i. We assume these parameters are externally provided; determining the appropriate parameter settings to meet application requirements is an important but separate problem, outside the scope of this paper. We also assume that the system includes an admission control component that ensures that the system capacity is adequate to serve the aggregate minimum reservations of all admitted clients. The behavior of the system if the assumption does not hold is discussed later in the section, along with alternative approaches.

Consider a simple setup with three VMs: one supporting remote desktop (RD), one running an Online Transaction Processing (OLTP) application and a Data Migration (DM) VM. The RD VM has a low throughput requirement but needs low IO latency for usability. OLTP runs a transaction processing workload requiring high throughput and low IO latency. The data migration workload requires high throughput but is insensitive to IO latency. Based on these requirements, the shares for RD, OLTP, and DM can be assigned as 100, 200, and 300 respectively. To provide low latency and a minimum degree of responsiveness, reservations of 250 IOPS each are specified for RD and OLTP. An upper limit of 1000 IOPS is set for the DM workload so that it cannot consume all the spare bandwidth in the system and cause high delays for the other workloads. The values chosen here are somewhat arbitrary, but were selected to highlight the use of various controls in a diverse workload scenario.

First consider how a conventional proportional scheduler would divide the total throughput T of the storage device. Since throughput is allocated to VMs in proportion to their weights, an active VM v_i will receive a throughput $T \times (w_i / \sum_j w_j)$, where the summation is over the weights of the active VMs (i.e. those with at least one pending IO). If the storage device's throughput is 1200 IOPS in the above example, RD will receive 200 IOPS, which is below its required minimum of 250 IOPS. This can lead to a poor experience for the RD user, even though there is sufficient system capacity for both RD and OLTP to receive their reservations of 250 IOPS. In our model, VMs always receive service between their minimum reservation and maximum limit (as long as system throughput is at least the aggregate of the reservations of active VMs).

In this case, mclock would provide RD with its minimum reservation of 250 IOPS and the remaining 950 IOPS would be divided between OLTP and DM in the ratio 2 : 3, resulting in allocations of 380 and 570 IOPS

respectively. Figure 3 shows the IOPS allocation to the three VMs in the example above, for different values of the system throughput, T. For T between 1500 and 2000 IOPS, the throughput is shared between RD, OLTP, and DM in proportion to their weights (1 : 2 : 3), since none of them will exceed their limit or fall below the reservation. If $T \geq 2000$ IOPS, then DM will be capped at 1000 IOPS because its share of $T/2$ is higher than its upper limit, and the remainder is divided between RD and OLTP in the ratio 1 : 2. If the total throughput T drops below 1500 IOPS, the allocation of RD bottoms out at 250 IOPS, and similarly at $T \leq 875$ IOPS, OLTP also bottoms out at 250 IOPS. Finally, for $T < 500$ IOPS, the reservations of RD and OLTP cannot be met; the available throughput will be divided equally between RD and OLTP (since their reservations are the same) and DM will receive no service. The last case should be rare if the admission controller estimates the overall throughput conservatively.

The allocation to a VM varies dynamically with the current throughput T and the set of active VMs. At any time, the VMs are partitioned into three sets: *reservation-clamped* ($\mathscr{R}$), *limit-clamped* ($\mathscr{L}$) or *proportional* ($\mathscr{P}$), based on whether their current allocation is clamped at the lower or upper bound or is in between. If T is the current throughput, we define $T_P = T - \sum_{j \in \mathscr{R}} r_j - \sum_{j \in \mathscr{L}} l_j$. The allocation γ_i made to active VM v_i for $T_P \geq 0$, is given by:

$$\gamma_i = \begin{cases} r_i & v_i \in \mathscr{R} \\ l_i & v_i \in \mathscr{L} \\ T_P \times (w_i / \sum_{j \in \mathscr{P}} w_j) & v_i \in \mathscr{P} \end{cases} \quad (1)$$

and

$$\sum_i \gamma_i = T. \quad (2)$$

When the system throughput T is known, the allocations γ_i can be computed explicitly. Such explicit computation is sometimes used for calculating CPU time allocations to virtual machines with service requirement specifications similar to these. When a VM exits or is powered on at the host, new service allocations are computed. In the case of a storage array, T is highly dependent on the presence of other hosts and the workload presented to the storage device. Since the throughput varies dynamically, the storage scheduler cannot rely upon service allocations computed at VM entry and exit times. The mClock scheduler ensures that the goals in Eq. (1) and (2) are satisfied continuously, even as the system's throughput varies, using a novel, lightweight tagging scheme.

Clearly, a feasible allocation is possible only if the aggregate reservation $\sum_j r_j$ does not exceed the total system throughput T. When $T_P < 0$, the system throughput is insufficient to meet the reservations; in this case mClock simply gives each VM throughput proportional to its reservation. This may not always be the desired behavior. VMs without a reservation may be starved in this case, but this problem can be easily avoided by adding a small default reservation for all VMs. In addition, one can add priority control to meet reservations based on priority levels. Exploring these options further is left to future work.

2.2 Proportional Share Algorithms

A number of approaches such as Stonehenge [23], SFQ(D) [24] and Argon [41] have been proposed for proportional sharing of storage between applications. Wang and Merchant [45] extended proportional sharing to distributed storage. Argon [41] and Aqua [48] propose service-time-based disk allocation to provide fairness as well as high efficiency. Brandt *et al.* [47] have proposed Hierarchical Disk Sharing, which uses hierarchical token buckets to provide isolation and bandwidth reservation among clients accessing the same disk. However, measuring per-request service times in our environment is difficult because multiple requests will typically be pending at the storage device.

Overall, none of these algorithms offers support for the combination of shares, reservations, and limits. Other methods for resource management in virtual clusters [16, 39] have been proposed, but they mainly focus on CPU and memory resources and do not address the challenges raised by variable capacity that *mClock* does.

2.3 Latency-sensitive Application Support

Several existing algorithms provide support for controlling the response time of latency-sensitive applications, but not strict latency guarantees or explicit latency targets. In the case of CPU scheduling, BVT [14], SMART [28], and lottery scheduling [37, 43] provide proportional allocation, latency-reducing mechanisms, and methods to handle priority inversion by exchanging tickets. Borrowed Virtual Time [14] and SMART [28] can give a short-term advantage to latency-sensitive applications by shifting their virtual tags relative to the other applications. pClock [22] and service-curve based methods [12, 27, 31, 36] decouple latency and throughput requirements, but like the other methods also do not support reservations and limits.

2.4 Reservation-Based Algorithms

For CPU scheduling and memory management, several approaches have been proposed for integrating reservations with proportional-share allocations [17, 34, 35]. In these models, clients either receive a *guaranteed fraction* of the server capacity (reservation-based clients) or a *share* (ratio) of the remaining capacity after satisfying

reservations (proportional-share-based clients). A standard proportional-share scheduler can be used in conjunction with an allocator that adjusts the weights of the active clients whenever there is a client arrival or departure. Guaranteeing minimum allocations for CPU time is relatively straightforward since its capacity (in terms of cycles/sec) is fixed and known, and allocating a given proportion would guarantee a certain minimum amount. The same idea does not apply to storage allocation where system throughput can fluctuate.

In our model the clients are not statically partitioned into reservation-based or proportional-share-based clients. Our model automatically modifies the entitlement of a client when service capacity changes due to changes in the workload characteristics or due to the arrival or departure of clients. The entitlement is at least equal to the reservation and can be higher if there is sufficient capacity. Since 2003, the VMware ESX Server has provided reservations and proportional-share controls for both CPU and memory resources in a commercial product [8, 42, 44]. These mechanisms support the same rich set of controls as in mClock, but do not handle varying service capacity.

Finally, operating system based frameworks like Rialto [25] provide fixed reservations for known-capacity CPU service, while allowing additional service requests to be honored on an availability basis. Rialto requires recomputation of an allocation graph on each new arrival, which is then used for CPU scheduling.

3 mClock Algorithm

Tag-based scheduling underlies many previously proposed fair-schedulers [10, 13, 15, 18]: all requests are assigned tags and scheduled in order of their tag values. For example, an algorithm can assign tags spaced by increments of $1/w_i$ to successive requests of client i; if all requests are scheduled in order of their tag values, the clients will receive service in proportion to w_i. In order to synchronize idle clients with the currently active ones, these algorithms also maintain a global tag value commonly known as *global virtual time* or just *virtual time*. In mClock, we extend this notion to use multiple tags based on three controls and dynamically decide which tag to use for scheduling, while still synchronizing idle clients.

The intuitive idea behind the mClock algorithm is to logically interleave a constraint-based scheduler and a weight-based scheduler in a fine-grained manner. The constraint-based scheduler ensures that VMs receive at least their minimum reserved service and no more than the upper limit in a time interval, while the weight-based scheduler allocates the remaining throughput to achieve proportional sharing. The scheduler alternates between phases during which one of these schedulers is active to maintain the desired allocation.

Symbol	Meaning
P_i^r	Share based tag of request r and VM v_i
R_i^r	Reservation tag of request r from v_i
L_i^r	Limit tag of request r from v_i
w_i	Weight of VM v_i
r_i	Reservation of VM v_i
l_i	Maximum service allowance (Limit) for v_i

Table 2: Symbols used and their descriptions

mClock uses two main ideas: *multiple real-time clocks* and *dynamic clock selection*. Each VM IO request is assigned three tags, one for each clock: a reservation tag R, a limit tag L, and a proportional share tag P for weight-based allocation. Different clocks are used to keep track of each of the three controls, and tags based on one of the clocks are dynamically chosen to do the constraint-based or weight-based scheduling.

The scheduler has three main components: (*i*) Tag Assignment (*ii*) Tag Adjustment and (*iii*) Request Scheduling. We will explain each of these in more detail below.

Tag Assignment: This routine assigns R, L and P tags to a request r from VM v_i arriving at time t. All the tags are assigned using the same underlying principle, which we illustrate here using the reservation tag. The R tag assigned to this request is the higher of the arrival time or the previous R tag $+ 1/r_i$. That is:

$$R_i^r = \max\{R_i^{r-1} + 1/r_i,\ \mathit{Current\ time}\} \qquad (3)$$

This gives us two key properties: first, the R tags of a continuously backlogged VM are spaced $1/r_i$ apart. In an interval of length T, a backlogged VM will have about $T \times r_i$ requests with R tag values in that interval. Second, if the current time is larger than this value due to v_i becoming active after a period of inactivity, the request is assigned an R tag equal to the current time. Thus idle VMs do not gain any idle credit for future service.

Similarly, the L tag is set to the maximum of the current time and $(L_i^{r-1} + 1/l_i)$. The L tags of a backlogged VM are spaced out by $1/l_i$. Hence, if the L tag of the first pending request of a VM is less than the current time, it has received less than its upper limit at this time. A limit tag higher than the current time would indicate that the VM has received its limit and should not be scheduled. The proportional share tag P_i^r is also the larger of the arrival time of the request and $(P_i^{r-1} + 1/w_i)$ and subsequent backlogged requests are spaced by $1/w_i$.

Tag Adjustment: Tag adjustment is used to calibrate the proportional share tags against real time. This is required whenever an idle VM becomes active again. In virtual time based schedulers [10, 15] this synchronization is done using global virtual time. The initial P tag value of a freshly active VM is set to the current time,

but the spacing of P tags after that is determined by the relative weights of the VMs. After the VM has been active for some time, the P tag values become unrelated to real time. This can lead to starvation when a new VM becomes active, since the existing P tags are unrelated to the P tag of the new VM. Hence existing P tags are adjusted so that the smallest P tag matches the time of arrival of the new VM, while maintaining their relative spacing. In the implementation, when a VM is activated, we assign it an offset equal to the difference between the effective value of the smallest existing P tag and the current time. During scheduling, the offset is added to the P tag to obtain the effective P tag value. The relative ordering of existing P tags is not altered by this transformation; however, it ensures that the newly activated VMs compete fairly with existing VMs.

Algorithm 1: Components of mClock algorithm

```
Max_QueueDepth = 32;

RequestArrival (request r, time t, vm v_i)
begin
    if v_i was idle then
        /* Tag Adjustment */
        minPtag = minimum of all P tags;
        foreach active VM v_j do
            P_j^r -= minPtag - t;
    /* Tag Assignment */
    R_i^r = max{R_i^(r-1) + 1/r_i, t} /* Reservation tag */
    L_i^r = max{L_i^(r-1) + 1/l_i, t} /* Limit tag */
    P_i^r = max{P_i^(r-1) + 1/w_i, t} /* Shares tag */
    ScheduleRequest();
end

ScheduleRequest ()
begin
    if Active_IOs >= Max_QueueDepth then
        return;
    Let E be the set of requests with R tag <= t
    if E not empty then
        /* constraint-based scheduling */
        select IO request with minimum R tag from
        E
    else
        /* weight-based scheduling */
        Let E' be the set of requests with L tag <= t
        if E' not empty OR Active_IOs == 0 then
            select IO request with minimum P tag
            from E'
            /* Assuming request belong to VM v_k */
            Subtract 1/r_k from R tags of VM v_k
    if IO request selected != NULL then
        Active_IOs++;
end

RequestCompletion (request r, vm v_i)
  Active_IOs -- ;
  ScheduleRequest();
```

Request Scheduling: mClock needs to check three different tags to make its scheduling decision instead of a single tag in previous algorithms. As noted earlier, the scheduler alternates between constraint-based and weight-based phases. First, the scheduler checks if there are any eligible VMs with R tags no more than the current time. If so, the request with smallest R tag is dispatched for service. This is defined as the constraint-based phase. This phase ends (and the weight-based phase begins) at a scheduling instant when all the R tags exceed the current time.

During a weight-based phase, all VMs have received their reservations guaranteed up to the current time. The scheduler therefore allocates server capacity to achieve proportional service. It chooses the request with smallest P tag, but only from VMs which have not reached their limit (whose L tag is smaller than the current time). Whenever a request from VM v_i is scheduled in a weight-based phase, the R tags of the outstanding requests of v_i are decreased by $1/r_i$. This maintains the condition that R tags are always spaced apart by $1/r_i$, so that reserved service is not affected by the service provided in the weight-based phase. Algorithm 1 provides pseudo code of various components of mClock.

3.1 Storage-specific Issues

There are several storage-specific issues that an IO scheduler needs to handle: IO bursts, request types, IO size, locality of requests and reservation settings.

Burst Handling. Storage workloads are known to be bursty, and requests from the same VM often have a high spatial locality. We help bursty workloads that were idle to gain a limited preference in scheduling when the system next has spare capacity. This is similar to some of the ideas proposed in BVT [14] and SMART [28]. However, we do it in a manner so that reservations are not impacted.

To accomplish this, we allow VMs to gain *idle credits*. In particular, when an idle VM becomes active, we compare the previous P tag with current time t and allow it to lag behind t by a bounded amount based on a VM-specific burst parameter. Instead of setting the P tag to the current time, we set it equal to $t - \sigma_i * (1/w_i)$. Hence the actual assignment looks like:

$$P_i^r = \max\{P_i^{r-1} + 1/w_i,\ t - \sigma_i/w_i\}$$

The parameter σ_i can be specified per VM and determines the maximum amount of credit that can be gained by becoming idle. Note that adjusting only the P tag

has the nice property that *it does not affect the reservations of other VMs*; however if there is spare capacity in the system, it will be preferentially given to the VM that was idle. This is because the R and L tags have strict priority over the P tags, so adjusting P tags cannot affect the constraint-based phase of the scheduler.

Request Type. mClock treats reads and writes identically. In practice writes show lower latency due to write buffering in the disk array. However doing any re-ordering of reads before writes for a single VM can lead to an inconsistent state of the virtual disk on a crash. Hence mClock schedules all IOs within a VM in a FCFS order without distinguishing between reads and writes.

IO size. Since larger IO sizes take longer to complete, differently-sized IOs should not be treated equally by the IO scheduler. We propose a technique to handle large-sized IOs during tagging. The IO latency with n random outstanding IOs with an IO size of S each can be written as:

$$Lat = n(T_m + S/B_{peak}) \quad (4)$$

Here T_m denotes the mechanical delay due to seek and disk rotation and B_{peak} denotes the peak transfer bandwidth of a disk. Converting the latency observed for an IO of size S_1 to an IO of a reference size S_2, keeping other factors constant would give:

$$Lat_2 = Lat_1 * (1 + \frac{S_2}{T_m \times B_{peak}})/(1 + \frac{S_1}{T_m \times B_{peak}}) \quad (5)$$

For a small reference IO size of $8KB$ and using typical values for mechanical delay T_m = $5ms$ and peak transfer rate, B_{peak} = 60 MB/s, the numerator = Lat_1*(1 + $8/300$) $\approx$ Lat_1. So, for tagging purposes, a single request of IO size S is treated as equivalent to: $(1 + S/(T_m \times B_{peak}))$ IO requests.

Request Location. mClock can detect sequentiality within a VM's workload, but in most virtualized environments the IO stream seen by the underlying storage may not be sequential due to a high degree of multiplexing. mClock improves the overall efficiency of the system by scheduling IOs with high locality as a batch. A VM is allowed to issue IO requests in a batch as long as the requests are close in logical block number space (e.g., within 4 MB). Also the size of batch is bounded by a configurable parameter (set to 8).

This optimization impacts the time granularity over which reservations are met. The batching of IOs is limited to a small number, typically 8. so for N VMs, the delay in meeting reservations can be $8N$ IOs. A typical number of VMs/host is 10-15, so this can delay reservation guarantees in the short term by the time taken to do roughly 100 IOs. Note that the benefit of batching and improved efficiency is distributed among all the VMs instead of giving it just to the VM with high sequentiality. It may be preferable to allocate the benefit of locality to the concerned VM; this is deferred to future work.

Reservation Setting. Admission control is a well known and difficult problem for storage devices due to their stateful nature and dependence of the throughput on the workload. We propose the simple approach of using the worst case IOPS from a storage device as an upper bound on sum of reservations for admission control. For example, an enterprise FC disk can service 200 to 250 random IOPS and a SATA disk can do roughly 80-100 IOPS. Based on the number and type of disk drives backing a storage LUN, one can obtain a conservative estimate of reservable throughput. This is what we have used to set parameters in our experiments. Also in order to set the reservations to meet an application's latency for a certain number of outstanding IOs, we use Little's law:

$$IOPS = \text{Outstanding IOs}/Latency \quad (6)$$

Thus, for an application that typically keeps 8 IOs outstanding and requires 25 ms average latency, the reservation should be set to 8 / 0.025 = 320 IOPS.

3.2 Distributed mClock

Cluster-based storage systems are emerging as a cost-effective, scalable alternative to expensive, centralized disk arrays. By using commodity hardware (both hosts and disks) and using software to glue together the storage distributed across the cluster, these systems allow for lower cost and more flexible provisioning than conventional disk arrays. The software can be designed to compensate for the reliability and consistency issues introduced by the distributed components.

Several research prototypes (e.g., CMU's Ursa Minor [9], HP Labs' FAB [30], IBM's Intelligent Bricks [46]) have been built, and several companies (such as LeftHand [4], Seanodes [6]) are offering iSCSI-based storage devices using local disks at virtualized hosts. In this section, we extend *mClock* to run on each storage server, with minimal communication between the servers, and yet provide per-VM globally (cluster-wide) proportional service, reservations, and limits.

3.2.1 dmClock Algorithm

dmClock runs a modified version of *mClock* at each server. There is only one modification to the algorithm to account for the distributed model in the Tag-Assignment component. During tag assignment each server needs to determine two things: the aggregate service received by the VM from all the servers in the system and the amount of service that was done as part of reservation. This information will be provided implicitly by the host running a VM by piggybacking two integers ρ_i and δ_i with each request that it forwards to a storage server s_j. Here δ_i denotes number of IO requests from VM v_i that have

completed service at all the servers between the previous request (from v_i) to the server s_j and the current request. Similarly, ρ_i denotes the number of IO requests from v_i that have been served as part of constraint-based phase between the previous request to s_j and the current request. This information can be easily maintained by the host running the VM. The host forwards the values of ρ_i and δ_i along with v_i's request to a server. (Note that for the single server case, ρ and δ will always be 1.) In the Tag-Assignment routine, these values are used to compute the tags as follows:

$$\begin{aligned} R_i^r &= \max\{R_i^{r-1} + \rho_i/r_i,\ t\} \\ L_i^r &= \max\{L_i^{r-1} + \delta_i/l_i,\ t\} \\ P_i^r &= \max\{P_i^{r-1} + \delta_i/w_i,\ t\} \end{aligned}$$

Hence, the new request may receive a tag further into the future, to reflect the fact that v_i has received additional service at other servers. The greater the value of δ, the lower the priority the request has for service. Note that this does not require any synchronization among the storage servers. The remainder of the algorithm remains unchanged. The values of ρ and δ may, in the worst case, be inaccurate by up to 1 request at each of the other servers. However, the dmClock algorithm does not require complex synchronization between the servers [32].

4 Performance Evaluation

In this section, we present results from a detailed evaluation of *mClock* using a prototype implementation in the VMware ESX server hypervisor [7, 40]. The changes required were small: the overall implementation took roughly 200 lines of C code in order to modify an existing scheduling framework. The resulting scheduler is lightweight, which is important because it is on the critical path for IO issues and completions. We examine the following key questions about mClock:

(1) Why is *mClock* needed? (2) Can *mClock* allocate service in proportion to weights, while meeting the reservation and limit constraints? (3) Can *mClock* handle bursts effectively and reduce latency by giving idle credit? (4) How effective is *dmClock* in providing isolation among dynamic workloads in a distributed storage environment?

4.1 Experimental Setup

We implemented mClock by modifying the SCSI scheduling layer in the IO stack of VMware ESX server hypervisor to construct our prototype. The ESX host was a Dell Poweredge 2950 server with 2 Intel Xeon 3.0 GHz dual-core processors, 8GB of RAM and two Qlogic HBAs connected to an EMC CLARiiON CX3-40 storage array over FC SAN. We used two different storage volumes: one hosted on a 10 disk RAID 0 disk group and another on a 10 disk, RAID 5 disk group. The host was configured to keep 32 IOs pending per LUN at the array, which is the default setting.

We used a diverse set of workloads, using different operating systems, workload generators, and configurations, to verify that mClock is robust under a variety of conditions. We used two kinds of VMs: (1) Linux (RHEL) VMs, each with a 10GB virtual disk, one VCPU and 512 MB memory, and (2) Windows server 2003 VMs, each with a 16GB virtual disk, one VCPU and 1 GB of memory. The disks hosting the operating systems for VMs were on a different storage LUN.

Three parameters were configured for each VM: a minimum reservation r_i IOPS, a global weight w_i, and maximum limit l_i IOPS. The workloads were generated using Iometer [5] in the Windows server VMs and our own micro-workload generator in the Linux RHEL VMs. For both cases, the workloads were specified using IO sizes, the percentage of reads, the percentage of random IOs, and the number of concurrent IOs. We used 32 concurrent IOs per workload in all experiments, unless otherwise stated. In addition to these micro-benchmark workloads, we used macro-benchmark workloads generated using Filebench [26].

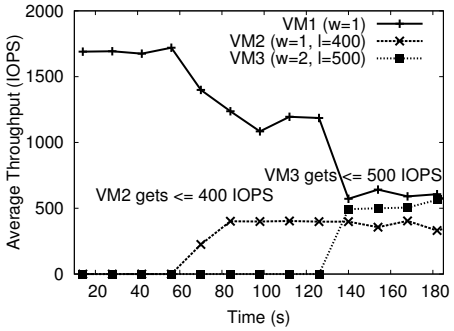
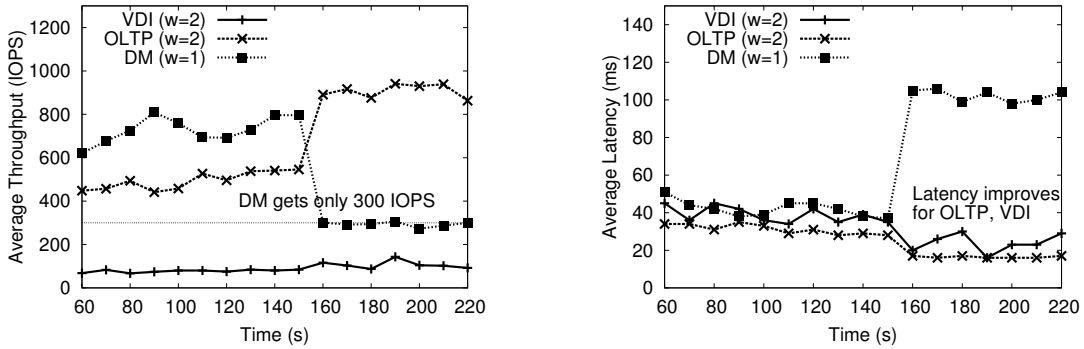


Figure 5: mClock limits the throughput of VM2 and VM3 to 400 and 500 IOPS as desired.

4.1.1 Limit Enforcement

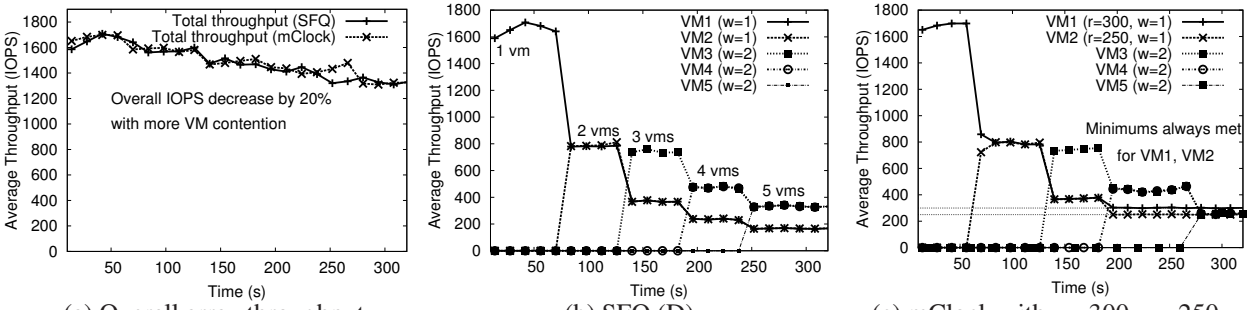
First we show the need for the limit control by demonstrating that pure proportional sharing cannot guarantee the specified number of IOPS and latency to a VM. We experimented with three workloads similar to those in the example of Section 2: RD, OLTP and DM.

RD is a bursty workload sending 32 random IOs (75% reads) of 4KB size every 250 ms. OLTP sends 8KB random IOs, 75% reads, and keeps 16 IOs pending at all times. The data migration workload DM does 32KB sequential reads, and keeps 32 IOs pending at all times. RD and OLTP are latency-sensitive workloads, requiring a response time under 30ms, while DM is not sensitive to latency. Accordingly, we set the weights in the ratio



(a) Throughput (b) Average Latency

Figure 4: Average throughput and latency for RD, OLTP and DM workloads, with weights = 2:2:1. At t=140 the limit for DM is set to 300 IOPS. mClock is able to restrict the DM workload to 300 IOPS and improve the latency of RD and OLTP workloads.



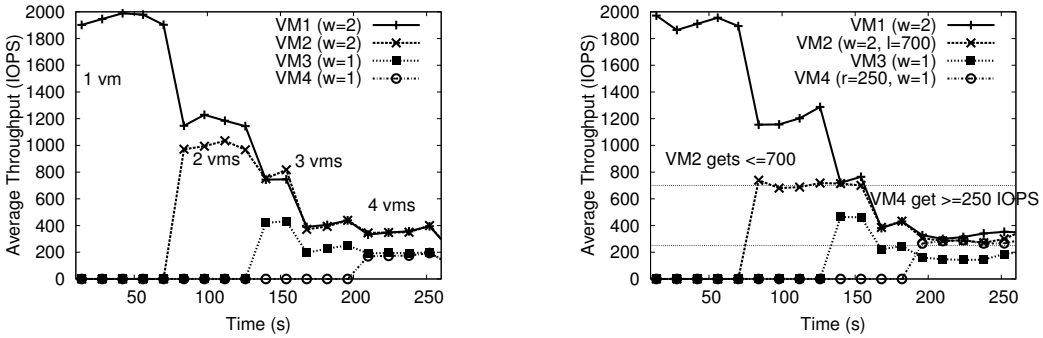
(a) Overall array throughput (b) SFQ (D) (c) mClock with r_1=300, r_2=250

Figure 6: Five VMs with weights in ratio 1:1:2:2:2. VMs are started at 60 sec intervals. The overall throughput decreases as more VMs are added. mClock enforces reservations and SFQ only does proportional allocation.

2:2:1 for the RD, OLTP, and DM workloads. First, we ran them with zero reservations and no limits in mClock, which is equivalent to running them with a standard fair scheduler such as SFQ(D) [24]. The throughput and latency achieved is shown in Figures 4(a) and (b), between times 60 and 140sec. Since RD was not fully backlogged, and OLTP had only 16 concurrent IOs, the work-conserving scheduler gave all the remaining queue slots (16 of them) to the DM workload. As a result, RD and OLTP got less than the specified proportion of IO throughput, while DM received more. Since the device queue was always heavily occupied by IO requests from DM, the latency seen by RD and OLTP was higher than desirable. We also experimented with other weight ratios (which are not shown here for lack of space), but saw no significant improvement, because the primary cause of the poor performance seen by RD and OLTP was that there were too many IOs from DM in the device queue.

To provide better throughput and lower latency to RD and OLTP workloads, we changed the upper limit for DM to 300 IOs (from unlimited) at $t = 140sec$. This caused the OLTP workload to see a 100% increase in throughput and the latency was reduced by half (36 ms to 16 ms). The RD workload also saw lower latency, while its throughput remained equal to its demand. This result shows that using limits with proportional sharing can be quite effective in reducing contention for critical workloads, and this effect cannot be produced using proportional sharing alone.

Next, we did an experiment to show that mClock effectively enforces limits in a more dynamic setting with workloads arriving at different times. Using Iometer on Windows Server VMs, we ran three workloads (VM1, VM2, and VM3), each generating 16KB random reads. We set the weights in the ratio 1:1:2, with limits of 400 IOPS on VM2 and 500 IOPS on VM3. We began with just VM1 and a new workload was started every 60 seconds. The storage device had a capacity of about 1600 random reads per second. Without the limits and based on the weights alone, we would expect the applications to receive 800 IOPS each when VM1 and VM2 are running, and 400, 400, and 800 IOPS respectively when



(a) SFQ(D) (b) mClock

Figure 7: Average throughput for VMs using SFQ(D) and mClock. mClock is able to restrict the allocation of VM2 to 700 IOPS and always provide at least 250 IOPS to VM4.

VM1, VM2, and VM3 are running together.

Figure 5 shows the throughput obtained by each of the workloads. When we added the VM2 (at time 60sec), it received only 400 IOPS based on its limit, and not the 800 IOPS it would have received based on the weights alone. When we started VM3 (at time 120sec), it received only its maximum limit, 500 IOPS, again smaller than its throughput share based on the weights alone. This shows that mClock is able to limit the throughput of VMs based on specified upper limits.

4.1.2 Reservations Enforcement

To test the ability of *mClock* to enforce reservations, we used a combination of 5 workloads, VM1 – VM5, all generated using Iometer on Windows Server VMs. Each workload maintained 32 outstanding IOs, all 16 KB random reads, at all times. We set their shares to the ratio 1:1:2:2:2. VM1 required a minimum of 300 IOPS, VM2 required 250 IOPS, and the rest had no minimum requirement. To demonstrate again the working of mClock in a dynamic environment, we began with just VM1, and a new workload was started every 60 seconds.

Figures 6(a) shows the overall throughput observed by the host using SFQ(D=32) and mClock. As the number of workloads increased, the overall throughput from the array decreased because the combined workload spanned larger numbers of tracks on the disks. Figures 6(b) and (c) show the throughput obtained by each workload using SFQ(D=32) and mClock respectively. When we used SFQ(D), the throughput of each VM decreased with increasing load, down to 160 IOPS for VM1 and VM2, while the remaining VMs received around 320 IOPS. In contrast, mClock provided 300 IOPS to VM1 and 250 IOPS to VM2, as desired. Increasing the throughput allocation also led to a smaller latency (as expected) for VM1 and VM2, which would not have been possible just using proportional shares.

VM	size, read%, random%	r_i	l_i	w_i
VM1	4K, 75%, 100%	0	MAX	2
VM2	8K, 90%, 80%	0	700	2
VM3	16K, 75%, 20%	0	MAX	1
VM4	8K, 50%,60%	250	MAX	1

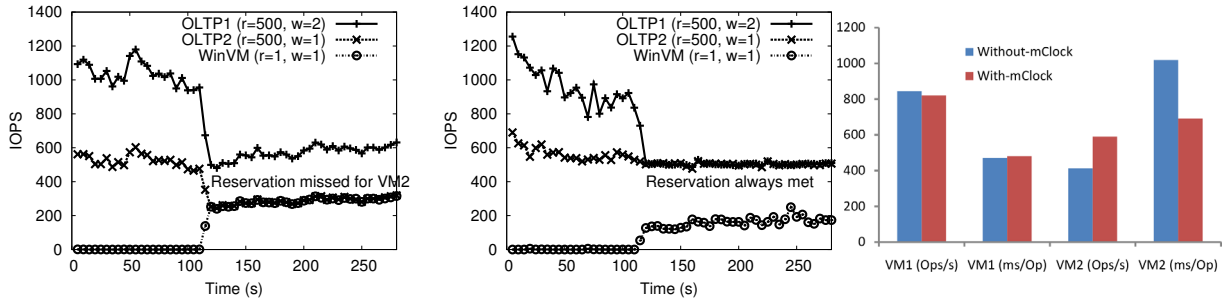
Table 3: VM workloads characteristics and parameters

4.1.3 Diverse VM Workloads

In the experiments above, we used mostly homogeneous workloads for ease of exposition and understanding. To demonstrate the effectiveness of mClock with a non-homogeneous combination of workloads, we experimented with workloads having very different IO characteristics. We used four workloads, generated using Iometer on Windows VMs, each keeping 32 IOs pending at all times. The workload configurations and the resource control settings (reservations, limits, and weights) are shown in Table 3.

Figures 7(a) and (b) show the throughputs allocated by SFQ(D) (weight-based allocation) and by mClock for these workloads. mClock was able to restrict VM2 to 700 IOPS, as desired, when only two VMs were doing IOs. Later, when VM4 became active, mClock was able to meet the reservation of 250 IOPS for it, whereas SFQ only provided around 190 IOPS. While meeting these constraints, mClock was able to keep the allocation in proportion to the weights of the VMs; for example, VM1 got twice as many IOPS as VM3 did.

We next used the same workloads to demonstrate how an administrator may determine the reservation to use. If the maximum latency desired and the maximum concurrency of the application is known, then the reservation can be simply estimated using Little's law as the ratio of the concurrency to the desired latency. In our case, if it is desired that the latency not exceed 65ms, the reservation can be computed as $32/0.065 = 492$, since the number of concurrent IOs from each application is 32. First, we



(a) Without mClock (b) With mClock (c)Application metrics

Figure 8: (a) Without mClock, VM2 missed its minimum requirement when WinVM started (b) With mClock, both OLTP workloads got their reserved IOPS despite WinVM workload (c) Application-level metrics: ops/s, avg Latency

VM	w_i	r_i=1, [IOPS, ms]	r_i=512, [IOPS,ms]
VM1	1	330, 96ms	**490**, 68ms
VM2	1	390, 82ms	**496**, 64ms
VM3	2	660, 48ms	**514**, 64ms
VM4	2	665, 48ms	**530**, 65ms

Table 4: mClock provided low latencies to VM1 and VM2 and throughputs close to the reservation when the reservations were changed from $r_i = 1$ to 512 IOPS.

ran the four VMs together with a reservation r_i = 1 each, and weights in the ratio 1:1:2:2.

The throughput (IOPS) and latency received by each in this simultaneous run are shown in Table 4. Note that workloads received IOPS in proportion to their weights, but the latencies of VM1 and VM2 were much higher than desired. We then set the reservation (r_i) for each VM to be 512 IOPS; the results are shown in the last column of Table 4. Note that first two VMs received higher IOPS of around 500 instead of 330 and 390, which is close to their reservation targets. The latency is also close to the expected value of 65ms. The other VMs saw a corresponding decline in their throughput. The reservation targets of VM1 and VM2 were not entirely met because the overall throughput was slightly smaller than the sum of reservations. This experiment demonstrates that mClock is able to provide a strong control to storage admins to meet their IOPS and latency targets for a given VM.

4.1.4 Bursty VM Workloads

Next, we experimented with the use of idle credits given to a workload for handling bursts. Recall that idle credits allow a workload to receive service in a burst only if the workload has been idle in the past and the reservations for all VMs have been met. This ensures that if an application is idle for a while, it gets preference when next there is spare capacity in the system. In this experiment, we used two workloads generated with Iometer on Windows Server VMs. The first workload was bursty, generating 128 IOs every 400ms, all 4KB reads, 80% random. The second was steady, producing 16 KB reads, 20% of them random and the rest sequential, with 32 outstanding IOs. Both VMs had equal shares, no reservation, and no limit imposed on the throughput. We used idle-credit (σ) values of 1 and 64 for our experiment.

VM	**σ=1, [IOPS, ms]**	**σ=64, [IOPS,ms]**
VM1	312, **49ms**	316, **30.8ms**
VM2	2420, 13.2ms	2460, 12.9ms

Table 5: The bursty workload (VM1) saw an improved latency when given a higher idle credit of 64. The overall throughput remained unaffected.

Table 5 shows the IOPS and average latency obtained by the bursty VM for the two settings of the idle credit. The number of IOPS were almost equal in either case because idle credits do not impact the overall bandwidth allocation over time, and VM1 had a bounded request rate. VM2 also saw almost the same IOPS for the two settings of idle credits. However, we notice that the latency seen by the bursty VM1 decreased as we increased the idle credits. VM2 also saw a similar or a slightly smaller latency, perhaps due to the increase in efficiency of doing several IOs at a time from a single VM, which are likely to be spatially closer on the storage device.

In the extreme, however, a very high setting of idle credits can lead to high latencies for non-bursty workloads by distorting the effect of the weights (although not the reservations or limits), and so we limit the setting to a maximum of 256 IOs in our implementation. This result indicates that using idle credits is an effective mechanism to help lower the latency of bursts.

4.1.5 Filebench Workloads

To test mClock with more realistic workloads, we experimented with two Linux RHEL VMs running OLTP workload using Filebench [26]. Each VMs was config-

ured with 1 VCPU, 512 MB of RAM, 10GB database disk, and 1 GB log virtual disk. To introduce throughput fluctuation another Windows 2003 VM running Iometer was used. The Iometer workload produced 32 concurrent, 16KB random reads. We assigned the weights in the ratio 2:1:1 to the two OLTP workloads and the Iometer workload, respectively, and gave a reservation of 500 IOPS to each OLTP workload. We initially started the two OLTP workloads together and then the Iometer workload at $t = 115s$.

Figures 8(a) and (b) show the IOPS received by the three workloads as measured inside the hypervisor, with and without mClock. Without mClock, as soon as the Iometer workload started, OLTP2 started missing its reservation and received around 250 IOPS. When run with mClock, both the OLTP workloads were able to achieve their reservations of 500 IOPS. This shows that mClock can protect critical workloads from a sudden change in the available throughput. The application-level metrics — the number of operations/sec and the transaction latency reported by Filebench — are summarized in Figure 8(c). Note that mClock was able to provide higher operations/sec and lower latency per operation in OLTP VMs, even with an increase in the overall IO contention.

4.2 dmClock Evaluation

In this section, we present results of a *dmClock* implementation in a distributed storage system. The system consisted of multiple storage servers (nodes) — three in our experiment. Each node was implemented using a virtual machine running RHEL Linux with a 10GB OS disk and a 10GB experimental disk, from which the data was served. Each experimental disk was placed on a different LUN backed by RAID-5 group with six disks. Thus, each experimental disk could do roughly 1500 IOPS for a random workload. A single storage device shared by all clients, was then constructed by striping across all the storage nodes. This configuration represents a clustered-storage system where there are multiple storage nodes, each with dedicated LUNs used for servicing IOs.

We implemented *dmClock* as a user-space module in each server node. The module receives IO requests containing IO size, offset, type (read/write), the δ and ρ parameters, and data in the case of write requests. The module can keep up to 16 outstanding IOs (using 16 threads) to execute the requests, and the requests are scheduled on these threads using the *dmClock* algorithm. The clients were run on a separate physical machine. Each client generated an IO workload for one or more storage nodes and also acted as a gateway, piggybacking the δ and ρ values onto each request sent to the storage nodes. Each client workload consisted of

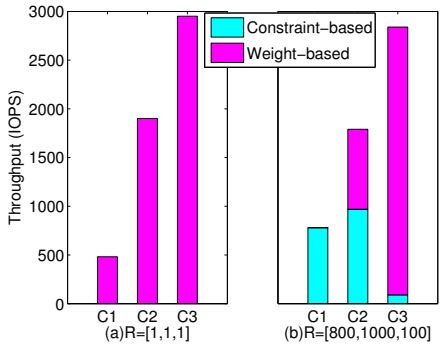


Figure 9: IOPS obtained by the three clients for two different cases. (a) All clients accessed the servers uniformly, with no reservations. (b) Clients had reservations of 800, 1000, and 100 IOPS, respectively.

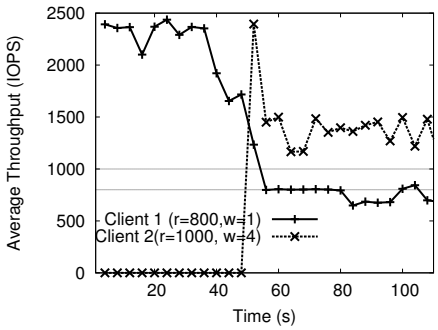


Figure 10: IOPS obtained by the two clients. When c_2 was started, c_1 still met its reservation target.

8KB random reads with 64 concurrent IOs, uniformly distributed over the nodes it used. We used our own workload generator here because of the need to add appropriate δ and ρ values to each request.

In first experiment, we used three clients, $\{c_1, c_2, c_3\}$, each accessing all three storage nodes. The weights were set in the ratio 1:4:6, with no upper limit on the IOPS. We experimented with two different cases: (1) No reservation per client, (2) Reservations of 800, 1000 and 100 for clients $\{c_1, c_2, c_3\}$ respectively. These values were used to highlight a use case where the allocation based on reservations may be higher than the allocation based on weights or shares for some clients. The output for these two cases is shown in Figure 9 (a) and (b). Case (a) shows the overall IO throughput obtained by three clients without reservations. As expected, each client received total service in proportion to its weight. In case (b), *dmClock* was able to meet the reservation goal of 800 IOPS for c_1, which would have been missed with a proportional share scheduler. The remaining throughput was divided between clients c_2 and c_3 in the ratio 2:3 as they respectively received around 1750 and 2700

IOPS. Figure 9(b) also shows the IOs done during the two phases of the algorithm.

Next, we experimented with non-uniform accesses from clients. In this case we used two clients c_1,c_2 and two storage servers. The reservations were set to 800 and 1000 IOPS and the weights were again in the ratio 1:4. c_1 sent IOs to the first storage node (S_1) only and we started c_2 after approximately 40 seconds. Figure 10 shows the IOPS obtained by the two clients with time. Initially, c_1 got the full capacity from server S_1 and when c_2 was started, c_1 was still able to get an allocation close to its reservation of 800 IOPS. The remaining capacity was allocated to c_2, which received around 1400 IOPS. A distributed weight-proportional scheduler [45] would have given approximately 440 IOPS to c_1 and the remainder to c_2, which would have missed the minimum requirement of c_1. This shows that even when the access pattern is non-uniform in a distributed environment, *dmClock* is able to meet reservations and assign overall IOPS in the ratio of weights to the extent possible.

5 Conclusions

In this paper, we presented a novel IO scheduling algorithm, *mClock*, that provides per-VM quality of service in presence of variable overall throughput. The QoS requirements for a VM are expressed as a minimum reservation, a maximum limit, and a proportional share. A key aspect of mClock is its ability to enforce such controls even with fluctuating overall capacity, as shown by our implementation in the VMware ESX server hypervisor. We also presented *dmClock,* a distributed version of our algorithm that can be used in clustered storage system architectures. We implemented *dmClock* in a distributed storage environment and showed that it works as specified, maintaining global per-client reservations, limits, and proportional shares, even though the schedulers run locally on the storage nodes.

The controls provided by *mClock* should allow stronger isolation between VMs. Although we have shown the effectiveness for hypervisor IO scheduling, we believe that the techniques are quite generic and can be applied to array-level scheduling and to other resources such as network bandwidth allocation as well. In our future work, we plan to explore further how to set these parameters to meet application-level SLAs.

6 Acknowledgement

We would like to thank our shepherd, Jon Howell, and the anonymous reviewers for their comments, which helped improve this paper. We thank Carl Waldspurger for valuable discussions and feedback on this work. Many thanks to Chethan Kumar for providing us with motivational use cases and Ganesha Shanmuganathan for discussions on the algorithm. Part of the work was done while the first author was a PhD student at Rice University [19]. The support of the National Science Foundation under Grants CNS-0541369 and CNS-0917157 is gratefully acknowledged. A preliminary version of the dmClock algorithm appeared as a brief announcement in PODC 2007 [21].

References

[1] Personal communications with many customers.

[2] Dell Inc. DVDStore benchmark. `http://delltechcenter.com/page/DVD+store`.

[3] Distributed Resource Scheduler, VMware Inc. `http://www.vmware.com/products/vi/vc/drs.html`.

[4] HP Lefthand SAN appliance. `http://www.lefthandsan.com/`.

[5] Iometer. `http://www.iometer.org`.

[6] Seanodes Inc. `http://www.seanodes.com/`.

[7] VMware ESX Server User Manual, December 2007. VMware Inc.

[8] vSphere Resource Management Guide, December 2009. VMware Inc.

[9] M. Abd-El-Malek et al. Ursa Minor: Versatile cluster-based storage. In *USENIX FAST*, 2005.

[10] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *INFOCOM*, pages 120–128, 1996.

[11] A. T. Clements, I. Ahmad, M. Vilayannur, J. Li, and V. Inc. Decentralized deduplication in SAN cluster file systems. In *USENIX Annual Technical Conference*, 2009.

[12] R. L. Cruz. Quality of service guarantees in virtual circuit switched networks. *IEEE Journal on Selected Areas in Communications*, 13(6):1048–1056, 1995.

[13] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. *Journal of Internetworking Research and Experience*, 1(1):3–26, September 1990.

[14] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. *SOSP:ACM Symposium on Operating Systems Principles*, 1999.

[15] S. Golestani. A self-clocked fair queueing scheme for broadband applications. In *INFOCOMM'94*, pages 636–646, April 1994.

[16] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. In *SOSP:ACM Symposium on Operating Systems Principles*, 1999.

[17] P. Goyal, X. Guo, and H. M. Vin. A hierarchial CPU scheduler for multimedia operating systems. *SIGOPS Oper. Syst. Rev.*, 30(SI):107–121, 1996.

[18] P. Goyal, H. M. Vin, and H. Cheng. Start-Time Fair Queuing: A scheduling algorithm for integrated services packet switching networks. Technical Report CS-TR-96-02, UT Austin, January 1996.

[19] A. Gulati. *Performance virtualization and QoS in Shared storage systems*. PhD thesis, Rice University, Houston, TX, USA, 2008.

[20] A. Gulati, I. Ahmad, and C. Waldspurger. PARDA: Proportional Allocation of Resources in Distributed Storage Access. In *(FAST '09)Proceedings of the Seventh Usenix Conference on File and Storage Technologies*, Feb 2009.

[21] A. Gulati, A. Merchant, and P. Varman. d-clock: Distributed QoS in heterogeneous resource environments. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 330–331, New York, NY, USA, 2007. ACM.

[22] A. Gulati, A. Merchant, and P. Varman. *p*Clock: An arrival curve based approach for QoS in shared storage systems. In *ACM SIGMETRICS*, 2007.

[23] L. Huang, G. Peng, and T. cker Chiueh. Multi-dimensional storage virtualization. In *ACM SIGMETRICS*, pages 14–24, 2004.

[24] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *ACM SIGMETRICS*, 2004.

[25] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *SOSP:ACM Symposium on Operating Systems Principles*, 1997.

[26] R. McDougall. Filebench: Application level file system benchmark. `http://www.solarisinternals.com/si/tools/filebench/index.php`.

[27] T. S. E. Ng, D. C. Stephens, I. Stoica, and H. Zhang. Supporting best-effort traffic with fair service curve. In *Measurement and Modeling of Computer Systems*, pages 218–219, 1999.

[28] J. Nieh and M. S. Lam. A smart scheduler for multimedia applications. *ACM Trans. Comput. Syst.*, 21(2):117–163, 2003.

[29] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw.*, 1(3):344–357, 1993.

[30] Y. Saito et al. FAB: building distributed enterprise disk arrays from commodity components. *SIGPLAN Not.*, 39(11):48–58, 2004.

[31] H. Sariowan, R. L. Cruz, and G. C. Polyzos. Scheduling for quality of service guarantees via service curves. In *Proceedings of the International Conference on Computer Communications and Networks*, pages 512–520, 1995.

[32] R. Stanojevic and R. Shorten. Fully decentralized emulation of best-effort and processor sharing queues. In *ACM SIGMETRICS*, 2008.

[33] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking*, 6(5):611–624, 1998.

[34] I. Stoica, H. Abdel-wahab, and K. Jeffay. On the duality between resource reservation and proportional share resource allocation. In *Proc. of Multimedia Computing and Networking*, pages 207–214, 1997.

[35] I. Stoica, H. Abdel-wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288–299, 1996.

[36] I. Stoica, H. Zhang, and T. S. E. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services. *IEEE/ACM Trans. Netw.*, 8(2):185–199, 2000.

[37] D. G. Sullivan and M. I. Seltzer. Isolation with Flexibility: A resource management framework for central servers. In *USENIX Annual Technical Conference*, 2000.

[38] S. Suri, G. Varghese, and G. Chandramenon. Leap forward virtual clock: A new fair queueing scheme with guaranteed delay and throughput fairness. In *INFOCOMM'97*, April 1997.

[39] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *ASPLOS-VIII*, pages 181–192, New York, NY, USA, 1998. ACM.

[40] VMware, Inc. *Introduction to VMware Infrastructure*. 2007. http://www.vmware.com/support/pubs/.

[41] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *USENIX FAST*, Berkeley, CA, USA, 2007.

[42] C. Waldspurger. Personal Communications.

[43] C. A. Waldspurger. *Lottery and stride scheduling: flexible proportional-share resource management*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.

[44] C. A. Waldspurger. Memory resource management in VMware ESX server. In *(OSDI'02): Proceedings of the Fifth symposium on Operating systems Design and Implementation*, 2002.

[45] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *Usenix FAST*, February 2007.

[46] W. Wilcke et al. IBM intelligent bricks project — petabytes and beyond. *IBM Journal of Research and Development*, 50, 2006.

[47] J. C. Wu, S. Banachowski, and S. A. Brandt. Hierarchical disk sharing for multimedia systems. In *NOSSDAV*. ACM, 2005.

[48] J. C. Wu and S. A. Brandt. The design and implementation of Aqua: an adaptive quality of service aware object-based storage device. In *Proc. of IEEE/NASA MSST*, pages 209–218, May 2006.

[49] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. In *IEEE MASCOTS*, pages 135–142, 2005.

[50] L. Zhang. VirtualClock: A new traffic control algorithm for packet-switched networks. *ACM Trans. Comput. Syst.*, 9(2):101–124.

Virtualize Everything but Time

Timothy Broomhead Laurence Cremean Julien Ridoux Darryl Veitch
Center for Ultra-Broadband Information Networks (CUBIN)
Department of Electrical & Electronic Engineering, The University of Melbourne, Australia
{*t.broomhead, l.cremean*}*@ugrad.unimelb.edu.au,* {*jridoux, dveitch*}*@unimelb.edu.au*

Abstract

We propose a new timekeeping architecture for virtualized systems, in the context of Xen. Built upon a *feed-forward* based *RADclock* synchronization algorithm, it ensures that the clocks in each OS sharing the hardware derive from a single central clock in a resource effective way, and that this clock is both accurate and robust. A key advantage is simple, seamless VM migration with consistent time. In contrast, the current Xen approach for timekeeping behaves very poorly under live migration, posing a major problem for applications such as financial transactions, gaming, and network measurement, which are critically dependent on reliable timekeeping. We also provide a detailed examination of the HPET and Xen Clocksource counters. Results are validated using a hardware-supported testbed.

1 Introduction

Virtualization represents a major movement in the evolution of computer infrastructure. Its many benefits include allowing the consolidation of server infrastructure onto fewer hardware platforms, resulting in easier management and energy savings. Virtualization enables the seamless migration of running guest operating systems (guest OSs), which reduces reliance on dedicated hardware, and eases maintenance and failure recovery.

Timekeeping is a core service on computing platforms, and accurate and reliable timekeeping is important in many contexts including network measurement and high-speed trading in finance. Other applications where accurate timing is essential to maintain at all times, and where virtualization can be expected to be used either now or in the future, include distributed databases, financial transactions, and gaming servers. The emerging market of outsourced cloud computing also requires accurate timing to manage and correctly bill customers using virtualized systems.

Software clocks are based on local hardware (oscillators), corrected using synchronization algorithms communicating with reference clocks. For cost and convenience reasons, reference clocks are queried over a network.

Since a notion of universally shared absolute time is tied to physics, timekeeping poses particular problems for virtualization, as a tight 'real' connection must be maintained across the OSs sharing the hardware. Both timekeeping and timestamping rely heavily on hardware counters. Virtualization adds an extra layer between the hardware and the OSs, which creates additional resource contention, and increased latencies, that impact performance.

In this paper we propose a new timekeeping architecture for para-virtualized systems, in the context of Xen [1]. Using a hardware-supported testbed, we show how the current approach using the *Network Time Protocol* (NTP) system is inadequate, in particular for VM migration. We explain how the *feed-forward* based synchronization adopted by the *RADclock* [14] allows a *dependent clock* paradigm to be used and ensures that all OSs sharing the hardware share the same (accurate and robust) clock in a resource effective way. This results in robust and seamless live migration because each physical host machine has its own unique clock, with hardware specific state, which never migrates. Only a stateless clock-reading function migrates. We also provide a detailed examination and comparison of the HPET and Xen Clocksource counters.

Neither the idea of a dependent clock, nor the *RADclock* algorithm, are new. The key contribution here is to show how the feed-forward nature, and stateless clock read function, employed by the *RADclock*, are ideally suited to make the dependent clock approach actually work. In what is the first evaluation of *RADclock* in a virtualized context, we show in detail that the resulting solution is orders of magnitude better than the current state of the art in terms of both average and peak error follow-

ing disruptive events, in particular live migration. We have integrated our work into the *RADclock* [14] packages for Linux, which now support the architecture for Xen described here.

After providing necessary background in Section 2, we motivate our work in depth by demonstrating the inadequacies of the status quo in Section 3. Since hardware counters are key to timekeeping in general and to our solution in particular, Section 4 provides a detailed examination of the behavior of counters of key importance to Xen. Section 5 describes and evaluates our proposed timing architecture on a single physical host, and Section 6 deals with migration. We conclude in Section 7.

2 Background

We provide background on Xen, hardware counters, timekeeping, the *RADclock* and NTP clocks, and comparison methodology.

To the best of our knowledge there is no directly relevant peer-reviewed published work on timekeeping in virtualized systems. A valuable resource however is [21].

2.1 Para-Virtualization and Xen

All virtualization techniques rely on a *hypervisor*, which, in the case of Xen [1, 2, 9], is a minimal kernel with exclusive access to hardware devices. The hypervisor provides a layer of abstraction from physical hardware, and manages physical resources on behalf of the guests, ensuring isolation between them. We work within the *para-virtualization* paradigm, whereby **all** guest OS's are modified to have awareness of, and access to, the native hardware via *hypercalls* to the hypervisor, which are similar to a system call. It is more challenging to support accurate timing under the alternative fully hardware virtualized paradigm [2], and we do not consider this here.

Although we work in the context of para-virtualized Xen, the architecture we propose has broader applicability. We focus on Linux OS's as this is the most active platform for Xen currently. In Xen, the guest OSs belong to two distinct categories: Dom0 and DomU. The former is a privileged system which has access to most hardware devices and provides virtual block and network devices for the other, DomU, guests.

2.2 Hardware Counters

The heart of any software clock is local oscillator hardware, accessed via dedicated counters. Counters commonly available today include the *Time Stamp Counter* (TSC) [6] which counts CPU cycles[1], the *Advanced Configuration and Power Interface* (ACPI) [10], and the *High Precision Event Timer* (HPET) [5].

The TSC enjoys high resolution and also very fast access. Care is however needed in architectures where a unique TSC of constant nominal frequency may not exist. This can occur because of multiple processors with unsynchronized TSC's, and/or power management effects resulting in stepped frequency changes and execution interruption. Such problems were endemic in architectures such as Intel Pentium III and IV, but have been resolved in recent architectures such as Intel Nehalem and AMD Barcelona.

In contrast to CPU counters like the TSC, HPET and ACPI are system-wide counters which are unaffected by processor speed issues. They are always on and run at constant nominal rate, unless the entire system is suspended, which we ignore here. HPET is accessed via a data bus and so has much slower access time than the TSC, as well as lower resolution as its nominal frequency is about 14.3MHz. ACPI has even lower resolution with a frequency of only 3.57MHz. It has even slower access, since it is also read via a bus but unlike HPET is not memory mapped.

Beyond the counter ticking itself, power management affects *all* counters through its impact on counter access latency, which naturally requires CPU instructions to be executed. Recent processors can, without stopping execution, move between different *P-States* where the operating frequency and/or voltage are varied to reduce energy consumption. Another strategy is to stop processor execution. Different such idle states, or *C-States* C0, C1, C2. . . are defined, where C0 is normal execution, and the deeper the state, the greater the latency penalty to wake from it [12]. The impact on latency of these strategies is discussed in more detail later.

2.3 Xen Clocksource

The Xen Clocksource is a hardware/software hybrid counter presented to guest OSs by the hypervisor. It aims to combine the reliability of a given *platform timer* (HPET here) with the low access latency of the TSC. It is based on using the TSC to interpolate between HPET readings made on 'ticks' of the periodic interrupt scheduling cycle of the OS (whose period is typically 1ms), and is scaled to a frequency of approximately 1GHz. It is a 64-bit cumulative counter, and is effectively initialized to zero for each guest when they boot (this is implemented by a 'system time' variable they keep) and monotonically increases.

The Xen Clocksource interpolation is a relatively complex mechanism that accounts for lost TSC ticks (it actively overwrites the TSC register) and frequency changes of the TSC due to power management (it main-

[1] TSC is x86 terminology, other architectures use other names.

tains a TSC scaling factor which can be used by guests to scale their TSC readings). Such compensation is needed on some older processors as described in Section 2.2.

2.4 Clock Fundamentals

A distinction must be drawn between the software clock itself, and timestamping. A clock may be perfect, yet timestamps made with it be very inaccurate due to large and/or variable access latency. Such timestamping errors will vary widely depending on context and may erroneously reflect on the clock itself.

By a *raw* timestamp we mean a reading of the underlying counter. For a clock C which reads $C(t)$ at true time t, the final timestamp will be a time in seconds based not only on the raw timestamp, but also the clock parameters set by the synchronization algorithm.

A local (scaled) counter is not a suitable clock and needs to be synchronized because all counters *drift* if left to themselves: their rate, although very close to constant (typically measured to 1 part in 10^6 or less), varies. Drift is primarily influenced by temperature.

Remote clock synchronization over a network is based on a (typically bidirectional) exchange of timing messages from an OS to a time server and back, giving rise to four timestamps: two made by the OS as the timing message (here an NTP packet) leaves then returns, and two made remotely by the time server. Typically, exchanges are made periodically: once every *poll-period*.

There are two key problems faced by remote synchronization. The first is to filter out the variability in the delays to and from the server, which effectively corrupt timestamps. This is the job of the clock synchronization algorithm, and it is judged on its ability to do this well (small error and small error variability) and consistently in real environments (robustness).

The second problem is that of a fundamental ambiguity between clock error and the degree of path asymmetry. Let $A = d^{\uparrow} - d^{\downarrow}$ denote the true path asymmetry, where $d^{\uparrow}$ and $d^{\downarrow}$ are the true minimum one-way delays to and from the server, respectively; and let $r = d^{\uparrow} + d^{\downarrow}$ be the minimal Round Trip Time (RTT). In the absence of any external side-information on A, we must guess a value, and $\hat{A} = 0$ is typically chosen, corresponding to a symmetric path. This allows the clock to be synchronized, but only up to an unknown additive error lying somewhere in the range $[-r, r]$. This ambiguity cannot be circumvented, even in principle, by any algorithm. We explore this further under Experimental Methodology below.

2.5 Synchronization Algorithms

The *ntpd* daemon [11] is the standard clock synchronization algorithm used today. It is a *feedback* based design, in particular since system clock timestamps are used to timestamp the timing packets. The existing kernel system clock, which provides the interface for user and kernel timestamping and which is *ntpd*-oriented, is disciplined by *ntpd*. The final software clock is therefore quite a complex system as the system clock has its own dynamics, which interacts with that of *ntpd* via feedback. On Xen, *ntpd* relies on the Xen Clocksource as its underlying counter.

The *RADclock* [20] (Robust Absolute and Difference Clock) is a recently proposed alternative clock synchronization algorithm based on a *feed-forward* design. Here timing packets are timestamped using raw packet timestamps. The clock error is then estimated based on these and the server timestamps, and subtracted out when the clock is read. This is a feed-forward approach, since errors are corrected based on post-processing outputs, and these are not themselves fed back into the next round of inputs. In other words, the raw timestamps are independent of clock state. The 'system clock' is now stateless, simply returning a function of parameters maintained by the algorithm.

More concretely, the (absolute) *RADclock* is defined as $C_a(t) = N(t) \cdot \bar{p} + K - E(t)$, where $N(t)$ is the raw timestamp made at true time t, $\bar{p}$ is a stable estimate of average counter period, K is a constant which aligns the origin to the required timescale (such as UTC), and $E(t)$ is the current estimate of the error of the 'uncorrected clock' $N(t) \cdot \bar{p} + K$ which is removed when the clock is read. The parameters $\bar{p}$, K and E are maintained by the clock algorithm (see [20] for details of the algorithm itself). The clock reading function simply reads (or is passed) the raw timestamp N for the event of interest, fetches the clock parameters, and returns $C_a(t)$.

The *RADclock* can use any counter which satisfies basic requirements, namely that it be cumulative, does not roll over, and has reasonable stability. In this paper we provide results using both the HPET and Xen Clocksource.

2.6 Experimental Methodology

We give a brief description of the main elements of our methodology for evaluating clock performance. More details can be found in [15].

The basic setup is shown in Figure 1. It incorporates our own Stratum-1 NTP server on the LAN as the reference clock, synchronised via a GPS-corrected atomic clock. NTP timing packets flow between the server and the clocks in the host machine (two OS's each with two

clocks are shown in the figure) for synchronization purposes, typically with a poll period of 16 seconds. For evaluation purposes, a separate box sends and receives a flow of UDP packets (with period 2 seconds) to the host, acting as a set of timestamping 'opportunities' for the clocks under test. In this paper a separate stream was sent to each OS in the host, but for a given OS, all clocks timestamp the same UDP flow.

Based on timestamps of the arrival and departure of the UDP packets, the testbed allows two kinds of comparisons.

External: a specialized packet stamping 'DAG' card [4] timestamps packets just before they enter the NIC of the host machine. These can be compared to the timestamps for the same packets taken by the clocks inside the host. The advantage is an independent assessment; the disadvantage is that there is 'system' lying between the two timestamping events, which adds a 'system noise' to the error measurement.

Internal: clocks inside the same OS timestamp the packets back-to-back (thanks to our kernel modifications), so subtracting these allows the clocks to be compared. The advantage is the elimination of the system noise between the timestamps; the disadvantage is that differences between the clocks cannot be attributed to any specific clock.

The results appearing in this paper all use the external comparison, but internal comparisons were also used as a key tool in the process of investigation and validation.

As far as possible experiments are run concurrently so that clocks to be compared experience close to identical conditions. For example, clocks in the same OS share the very same NTP packets to the time server (and hence in particular, share the same poll period). There are a number of subtle issues we have addressed regarding the equivalence between what the test UDP packets, and the NTP packets actually used by the algorithm, 'see', which depends on details of the relative timestamping locations in the kernel. This topic is discussed further below in relation to 'host asymmetry'.

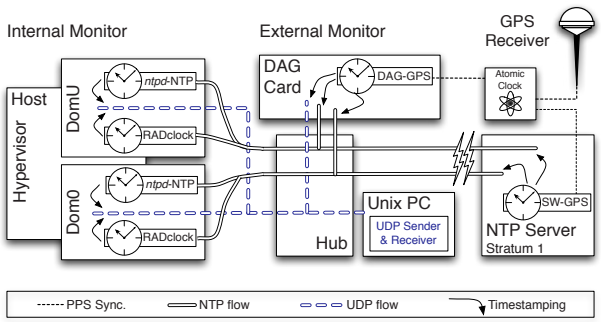


Figure 1: Testbed and clock comparison methodology.

It is essential to note that the delays experienced by timing packets have components both in the network, and in the host itself (namely the NIC + hardware + OS), each with their own minimum RTT and asymmetry values. Whereas the DAG card timestamps enable the network side to be independently measured and corrected, the same is not true of the host side component. Even when using the same server then, a comparison of different clocks, which have different asymmetry induced errors, is problematic. Although the spread of errors can be meaningfully compared, the median errors can only be compared up to some limit imposed by the (good but not perfect) methodology, which is of the order of 1 to 10 µs. Despite these limitations, we believe our methodology to be the best available at this time.

3 Inadequacy of the Status-Quo

The current timekeeping solution for Xen is built on top of the *ntpd* daemon. The single most important thing then regarding the performance of Xen timekeeping is to understand the behavior first of *ntpd* in general, and then in the virtualized environment.

There is no doubt that *ntpd* can perform well under the right conditions. If a good quality nearby time server is available, and if *ntpd* is well configured, then its performance on modern kernels is typically in the tens of microseconds range and can rival that of the *RADclock*. An example in the Xen context is provided in Figure 2, where the server is a Stratum-1 on the same LAN, and both *RADclock* and *ntpd*, running on Dom0 in parallel, are synchronizing to it using the same stream of NTP packets. Here we use the host machine **kultarr**, a 2.13GHz Intel Core 2 Duo. Xen selects a single CPU for use with Xen Clocksource, which is then used by *ntpd*. Power management is disabled in the BIOS.

The errors show a similar spread, with an Inter-Quartile Range (IQR) of around 10 µs for each. Note that here path asymmetry effects have not been accounted for, so that as discussed above the median errors do not reflect the exact median error for either clock.

In this paper our focus is on the right architecture for timing in virtualized systems, in particular such that seamless VM migration becomes simple and reliable, and not any performance appraisal of *ntpd* per se. Accordingly, unless stated otherwise we consistently adopt the configuration which maximizes *ntpd* performance – single nearby statically allocated Stratum-1 server, static and small polling period.

The problem with *ntpd* is the sudden performance degradations which can occur when conditions deviate from 'ideal'. We have detailed these robustness issues of *ntpd* in prior work, including [17, 18]. Simply put, when path delay variability exceeds some threshold, which is a

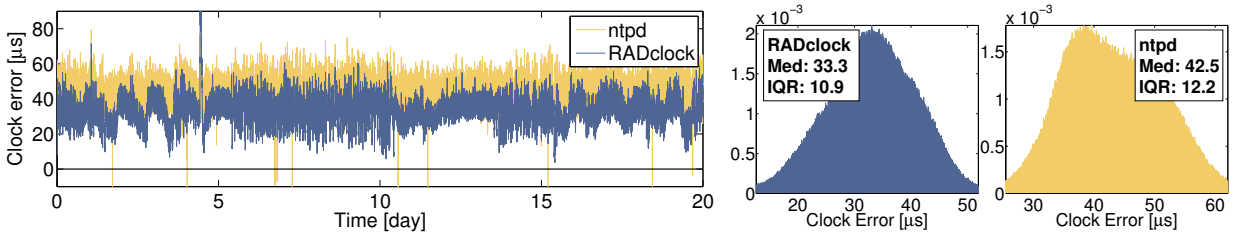


Figure 2: *RADclock* and *ntpd* uncorrected performance on dom0, measured using the external comparison with DAG.

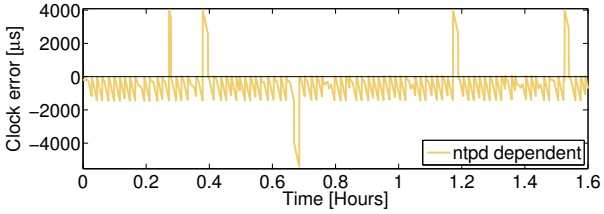


Figure 3: Error in the DomU dependent clock (dependent on the *ntpd* on Dom0 shown in Figure 2), measured using the external comparison with DAG. This 2 hour zoom is representative of an experiment 20 days long.

complex function of parameters, stability of the feedback control is lost, resulting in errors which can be large over small to very long periods. Recovery from such periods is also subject to long convergence times.

Consider now the pitfalls of using *ntpd* for timekeeping in Xen, through the following three scenarios.

Example 1 – Dependent *ntpd* clock In a *dependent* clock paradigm, only Dom0 runs a full clock synchronization algorithm, in this case *ntpd*. Here we use a 2.6.26 kernel, the last one supporting *ntpd* dependent timekeeping.

In the solution detailed in [19], synchronizing timestamps from *ntpd* are communicated to DomU guests via the periodic adjustment of a 'boot time' variable in the hypervisor. Timestamping in DomU is achieved by a modified system clock call which uses Xen Clocksource to extrapolate from the last time this variable was updated forward to the current time. The extrapolation assumes Xen Clocksource to be exactly 1GHz, resulting in a sawtooth shaped clock error which, in the example from our testbed given in Figure 3, is in the millisecond range. This is despite the fact that the Dom0 clock it derives from is the one depicted in Figure 2, which has excellent performance in the 10 µs range.

These large errors are ultimately a result of the way in which *ntpd* interacts with the system clock. With no *ntpd* running on DomU (the whole point of the dependent clock approach), the system clock has no way of intelligently correcting the drift of the underlying counter, in this case Xen Clocksource. The fact that Xen Clocksource is in reality only very approximately 1GHz means that this drift is rapid, indeed appearing to first order as a simple skew, that is a constant error in frequency. This failure of the dependent clock approach using *ntpd* has led to the alternative solution used today, where each guest runs its own independent *ntpd* daemon.

Example 2 – Independent *ntpd* clock In an *independent* clock paradigm, which is used currently in Xen timekeeping, each guest OS (both Dom0 and DomU) independently runs its own synchronization algorithm, in this case *ntpd*, which connects to its own server using its own flow of NTP timing packets. Clearly this solution is not ideal in terms of the frugal use of server, network, NIC and host resources. In terms of performance, the underlying issue is that the additional latencies suffered by guests in the virtual context make it more likely *ntpd* will be pushed into instability. Important examples of such latencies are the descheduling and time-multiplexing of guests across physical cores.

An example is given in Figure 4, where, despite synchronizing to the same high quality server on the LAN as before, stability is lost and errors reach the multiple millisecond range. This was brought about simply by adding a moderate amount of system load (some light churn of DomU guests and some moderate CPU activity on other guests), and allowing NTP to select its own polling period (in fact the default configuration), rather than fixing it to a constant value.

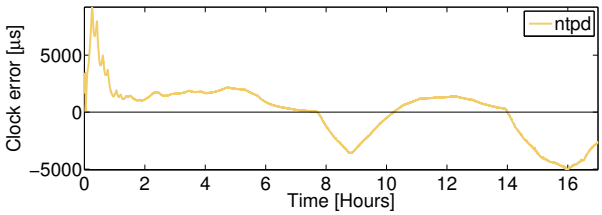


Figure 4: Error in the *ntpd* independent clock on DomU synchronizing to a Stratum-1 server on the LAN, with polling period set by *ntpd*. Additional guests are created and destroyed over time.

Example 3 – Migrating independent *ntpd* clock This example considers the impact of migration on the synchronization of the *ntpd* independent clock (the current solution) of a migrating guest. Since migration is treated in detail in Section 6, we restrict ourselves here to pointing to Figure 11, where extreme disruption – of the order of seconds – is seen following migration events. This is not a function of this particular example but is a generic result of the design of *ntpd* in conjunction with the independent clock paradigm. A dependent *ntpd* clock solution would not exhibit such behavior under migration, however it suffers from other problems as detailed above.

In summary, there are compelling design and robustness reasons for why *ntpd* is ill suited to timekeeping in virtual environments. The *RADclock* solution does not suffer from any of the drawbacks detailed above. It is highly robust to disruptions in general as described for example in [20, 15, 16], is naturally suited to a dependent clock paradigm as detailed in Section 5, as well as to migration (Section 6).

4 Performance of Xen Clocksource

The Xen Clocksource hybrid counter is a central component of the current timing solution under Xen. In this section we examine its access latency under different conditions. We also compare it to that of HPET, both because HPET is a core component of Xen Clocksource, so this enables us to better understand how the latter is performing, and because HPET is a good choice of counter, being widely available and uninfluenced by power management. This section also provides the detailed background necessary for subsequent discussions on network noise.

Access latency, which impacts directly on timekeeping, depends on the access mechanism. Since the timing architecture we propose in Section 5 is based on a feed-forward paradigm, to be of relevance our latency measurements must be of access mechanisms that are adequate to support feed-forward based synchronization. The fundamental requirement is that a counter be defined, which is cumulative, wide enough to not wrap between reboots (we use 64-bit counters which take 585 years to roll over on a 1GHz processor), and accessible from both kernel and user context. The existing *ntpd*-oriented software clock mechanisms do not satisfy these conditions. We describe below the alternatives we implement.

4.1 Baseline Latencies

In this section we use the host machine **kultarr**, a 2.13GHz Intel Core 2 Duo, and measure access latencies by counting the number of elapsed CPU cycles. For this purpose we use the *rdtsc()* function, a wrapper for the x86 RDTSC instruction to read the relevant register(s) containing the TSC value and to return it as a 64-bit integer. This provides direct access to the TSC with very low overhead from both user and kernel space. To ensure a unique and reliable TSC, from the BIOS we disable power management (both P-states and C-states), and also disable the second core to avoid any potential failure of TSC synchronization across the cores.

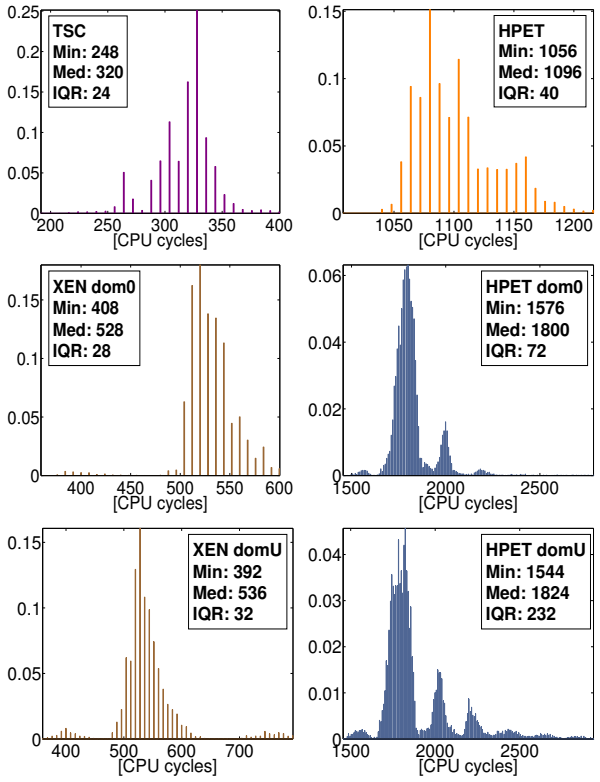


Figure 5: Distribution of counter latencies. Top: TSC and HPET in an unvirtualized system using feed-forward compatible access; Middle: Xen Clocksource and HPET from Dom0; Bottom: from DomU.

We begin by providing a benchmark result for HPET on a non-virtualized system. The top right plot in Figure 5 gives its latency histogram measured from within kernel context, using the access mechanism described in [3]. This mechanism augments the Linux *clocksource* code (which supports a choice among available hardware counters), to expose a 64-bit cumulative version of the selected counter. For comparison, in the top left plot we give the latency of the TSC accessed in the same way – it is much smaller as expected, but both counters have low variability. Note (see [3]) that the latency of TSC accessed directly via *rdtsc()* is only around 80 cycles, so that the feed-forward friendly access mechanism entails an overhead of around 240 cycles on this system.

Now consider latency in a Xen system. To provide the feed-forward compatible access required for each of HPET and Xen Clocksource, we first modified the Xen Hypervisor (4.0) and the Linux kernel 2.6.31.13 (Xen pvops branch) to expose the HPET to Dom0 and DomU. We added a new hypercall entry that retrieves the current raw platform timer value, HPET in this case. Like Xen Clocksource, this is a 64-bit counter which satisfies the cumulative requirement. We then added a system call to enable each counter to be accessed from user context. Finally, for our purposes here we added an additional system call that measures the latency of either HPET or Xen Clocksource from within kernel context using *rdtsc()*.

The middle row in Figure 5 shows the latency of Xen Clocksource and HPET from Dom0's kernel point of view. The Xen Clocksource interpolation mechanism adds an extra 200 CPU cycles compared to accessing the TSC alone in the manner seen above, for a total of 250 ns at this CPU frequency. The HPET latency suffers from the penalty created by the hypercall needed to access it, lifting its median value by 800 cycles for a total latency of 740 ns. More importantly, we see that the hypercall also adds more variability, with an IQR that increases from 40 to 72 CPU cycles, and the appearance of a multi-modal distribution which we speculate arises from some form of contention among hypercalls. The Xen Clocksource in comparison has an IQR only slightly larger than that of the TSC counter.

The bottom row in Figure 5 shows the latency of Xen Clocksource and HPET from DomU's kernel point of view. The Xen Clocksource shows the same performance as in the Dom0 case. HPET is affected more, with an increase in the number of modes and the mass within them, resulting in a considerably increased IQR.

In conclusion, Xen Clocksource performs well despite the overhead of its software interpolation scheme. In particular, although its latency is almost double that of a simple TSC access (and 7 times a native TSC access), it does not add a significant latency variability even when accessed from DomU. On the other hand however, the simple feed-forward compatible way of accessing HPET used here is only four times slower than the much more complicated Xen Clocksource and is still under 1 µs. This performance could certainly be improved, for example by replacing the hypercall by a dedicated mechanism such as a read-only memory-mapped interface.

4.2 Impact of Power Management

Power management is one of the key mechanisms potentially affecting timekeeping. The Xen Clocksource is designed to compensate for its effects in some respects. Here we examine its ultimate success in terms of latency.

In this section we use the host machine **sarigue**, a 3GHz Intel Core 2 Duo E8400. Since we do not use *rdtsc()* for latency measurement in this section, we do not disable the second core as we did before. Instead we measure time differences in seconds, to sub-nanosecond precision, using the *RADclock* difference clock [20] with HPET as the underlying counter[2]. P-states are disabled in the BIOS, but C-states are enabled.

Ideally one would like to directly measure Xen Clocksource's interpolation mechanism and so evaluate it in detail. However, the Xen Clocksource recalibrates the HPET interpolation on every change in P-State (frequency changes) or C-State (execution interruption), as well as once per second. Since oscillations for example between C-States occur hundreds of times per second [8], it is not possible to reliably timestamp these events, forcing us to look at coarser characterizations of performance.

From the timestamping perspective, the main problem is the obvious additional latency due to the computer being idle in a C-State when an event to timestamp occurs. For example, returning from C-State C3 to execution C0 takes about 20 µs [8].

For the purpose of synchronization over the network, the timestamping of outgoing and incoming synchronization packets is of particular interest. A useful measure of these is the Round-Trip-Time (RTT) of a request-reply exchange, however since this includes network queuing and delays at the time server as well as delays in the host, it is of limited use in isolating the latter.

To observe the host latency we introduce a metric we call the *RTThost*, which is roughly speaking the component of the RTT that lies within the host. More precisely, for a given request-reply packet pair, the RTThost is the sum of the two one-way delays from the host to the DAG card, and from the DAG card to the host. It is not possible to reliably measure these one-way delays individually in our testbed. However, the RTThost can be reliably measured as the difference of the RTT seen by the host and that seen by the DAG card. The RTThost is a measure of 'system noise' with a specific focus on packet timestamping. The smaller RTThost, the less noisy the host, and the higher the quality of packet timestamps.

Figure 6 shows RTThost values measured over 80 hours on two DomU guests on the same host. The capture starts with only the C-State C0 enabled, that is with power management functions disabled. Over the period of the capture, deeper C-States are progressively enabled and we observe the impact on RTThost. At each stage the CPU moves between the active state C0 and the idle states enabled at the time. Table 1 gives a breakdown of

[2]This level of precision relates to the difference clock itself, when measuring time differences of size of the order of 100 µs as here. It does not take into account the separate issue of timestamping errors, such as the (much larger!) counter access latencies studied above.

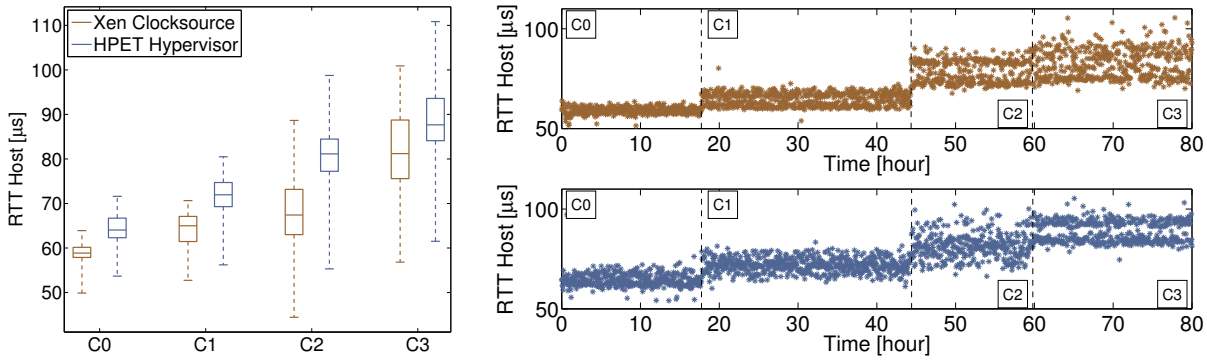


Figure 6: System noise as a function of the deepest enabled C-State for Xen Clocksource (upper time series and left box plots) and HPET (lower time series and right box plots). Time series plots have been sampled for clarity.

time spend in different states. It shows that typically the CPU will rest in the deepest allowed C-state unless there is a task to perform.

The left plot in Figure 6 is a compact representation of the distribution of RTThost values for Xen Clocksource and HPET, for each section of the corresponding time series presented on the right of the figure. Here whiskers show the minimum and 99th percentile values, the lower and upper sides of the box give the 25th and 75th percentiles, while the internal horizontal line marks the median.

The main observation is that, for each counter, RTThost generally increases with the number of C-States enabled, although it is slightly higher for HPET. The increase in median RTThost from C0 to C3 is about 20 µs, a value consistent with [8]. The minimum value is largely unaffected however, consistent with the fact that if a packet (which of course is sent when the host is in C0), is also received when it is in C0, then it would see the RTThost corresponding to C0, even if it went idle in between.

We saw earlier that the access latencies of HPET and Xen Clocksource differ by less than 1 µs, and so this cannot explain the differences in their RTThost median values seen here for each given C-State. These are in fact due to the slightly different packet processing in the two DomU systems.

	C0	C1	C2	C3
C0 enabled	100%	–	–	–
C1 enabled	2.17%	97.83%	–	–
C2 enabled	2.85%	0.00%	97.15%	–
C3 enabled	2.45%	0.00%	1.84%	95.71%

Table 1: Residency time in different C-States. Here "Cn enabled" denotes that all states from C0 up to Cn are enabled.

We conclude that Xen Clocksource, and HPET using our proof of concept access mechanism, are affected by power management when it comes to details of timestamping latency. These translate into timestamping errors, which will impact both clock reading and potentially clock synchronization itself. The final size of such errors however is also crucially dependent on the asymmetry value associated to RTThost, which is unknown. Thus the RTTHost measurements effectively place a bound on the system noise affecting timestamping, but do not determine it.

5 New Architecture for Virtualized Clocks

In this section we examine the performance and detail the benefits of the *RADclock* algorithm in the Xen environment, describe important packet timestamping issues which directly impact clock performance, and finally propose a new feed-forward based clock architecture for para-virtualized systems.

In Section 5.1 we use **sarigue**, and in Section 5.2 **kultarr**, with the same BIOS and power management settings described earlier.

5.1 Independent *RADclock* Performance

We begin with a look at the performance of the *RADclock* in a Xen environment. Figure 7 shows the final error of two independent *RADclock*s, one using HPET and the other Xen Clocksource, running concurrently in two different DomU guests. Separate NTP packet streams are used to the same Stratum-1 server on the LAN with a poll period of 16 seconds. The clock error for each clock has been corrected for path asymmetry, in order to reveal the underlying performance of the algorithm as a delay variability filter (this is possible in our testbed, but impossible for the clocks in normal operation). The difference of median values between the two clocks is

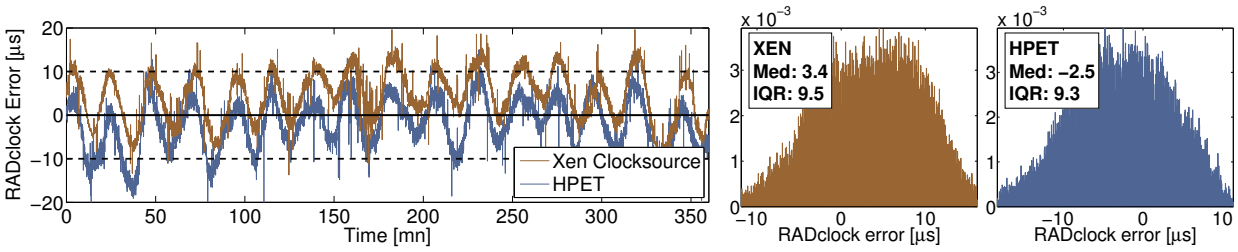


Figure 7: *RADclock* performance in state C0 using Xen Clocksource and HPET, running in parallel, each in a separate DomU guest.

extremely small, and below the detection level of our methodology. We conclude that the clocks essentially have identical median performance.

In terms of clock stability, as measured by the IQR of the clock errors, the two *RADclock* instances are again extremely similar, which reinforces our earlier observations that the difference in stability of the Xen Clocksource and HPET is very small (below the level of detection in our testbed), and that *RADclock* works well with any appropriate counter. The low frequency oscillation present in the time series here is due to the periodic cycle of the air conditioning system in the machine room, and affects both clocks in a similar manner consistent with previous results [3]. It is clearly responsible for the bulk of the *RADclock* error in this and other experiments shown in this paper.

Power management is also an important factor that may impact performance. Figure 8 shows the distribution of clock errors of the *RADclock*, again using HPET and the Xen Clocksource separately but concurrently as above, with different C-State levels enabled. In this case the median of each distribution has simply been shifted to zero to ease the stability (IQR) comparison. For each of the C-State levels shown, the stability of the *RADclock* is essentially unaffected by the choice of counter.

As shown in Figure 6, power management creates additional delays of higher variability when timestamping timing packets exchanged with the reference clock. The near indifference of the IQR given in Figure 8 to C-State shows that the *RADclock* filtering is robust enough to see through this extra noise.

Power management also has an impact on the asymmetry error all synchronization algorithms must face. In an excellent example of systematic observation bias, in a bidirectional paradigm a packet send by an OS would not be delayed by the power management strategy, because the OS chooses to enter an idle state only when it has nothing to do. On the other hand, over the time interval defined by the RTT of a time request, it is likely the host will choose to stop its execution and enter an idle state (perhaps a great many times) and the returning packet may find the system in such a state. Consequently, only the timestamping of received packets is likely to be affected by power management, which translates into a bias towards an extra path asymmetry, in the sense of 'most but not all packets', in the receiving direction. This bias is difficult to measure independently and authoritatively. The measurement of the RTThost shown in Figure 6 gives however a direct estimate of an upper bound for it.

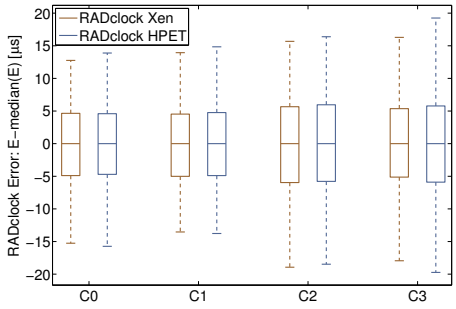


Figure 8: Compact centred distributions of *RADclock* performance as a function of the deepest C-State enabled (whiskers give 1st to 99th percentile).

5.2 Sharing the Network Card

The quality of the timestamping of network packets is crucial to the accuracy the synchronization algorithm can achieve. The networking in Xen relies on a firewall and networking bridge managed by Dom0. In Figure 9 we observe the impact of system load on the performance of this mechanism.

The top plot shows the RTThost time series, as seen from the Dom0 perspective, as we add more DomU guests to the host. Starting with Dom0 only, we add an additional guest every 12 hours. None of the OSs run any CPU or networking intensive tasks. The middle plot gives the box plots of the time series above, where the increase in median and IQR values is more clearly seen. For reference the 'native' RTThost of a non-virtualized system is also plotted. The jump from this distribution

to the one labeled 'Dom0' represents the cost of the networking bridge implemented in Xen.

The last plot in figure 9 shows the distribution of RTThost values from each guest's perspective. All guests have much worse performance than Dom0, but performance degrades by a similar amount as Dom0 as a function of the number of guests. For a given guest load level, the performance of each guest clock seems essentially the same, though with small systematic differences which may point to scheduling policies.

The observations above call for the design of a timestamping system under a dependent clock paradigm where Dom0 has an even higher priority in terms of networking, so that it can optimize its timestamping quality and thereby minimize the error in the central Dom0 clock, to the benefit of all clocks on the system. Further, DomU packet timestamping should be designed to minimize any differences between DomU guests, and reduce as much as possible the difference in host asymmetry between Dom0 and DomU guests, to help make the timestamping performance across the whole system more uniform.

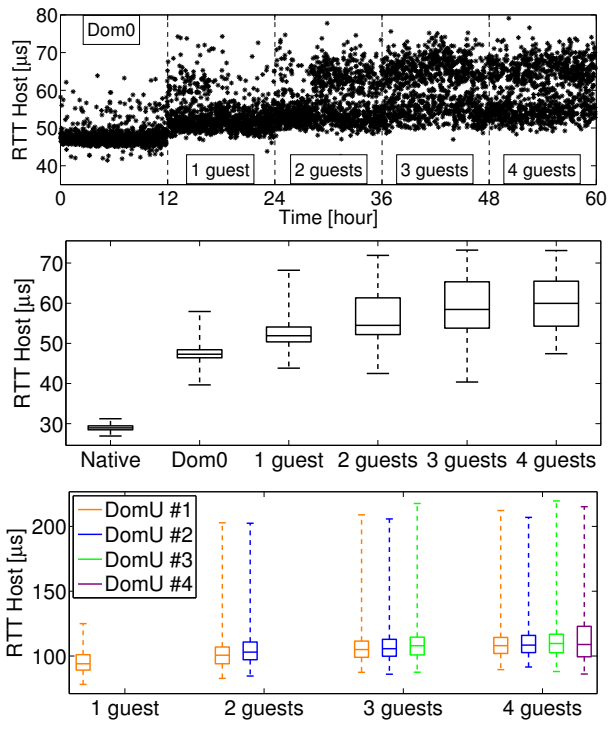


Figure 9: **kultarr**: RTThost (a.k.a. system noise) as a function of the number of active guests. Top: RTThost timeseries seen by Dom0; Middle: corresponding distribution summaries (with native non-Xen case added on the left for comparison); Bottom: as seen by each DomU. Whiskers show the minimum and 99th percentile.

5.3 A Feed-Forward Architecture

As described in Section 2.5, the feed-forward approach used by the *RADclock* has the advantage of cleanly separating timestamping (performed as a raw timestamp in the kernel or user space as needed), which is stateless, and the clock synchronization algorithm itself, which operates asynchronously in user space. The algorithm updates clock parameters and makes them available through the OS, where any authorized clock reading function (a kind of almost trivial stateless 'system clock') can pick them up and use them either to compose an absolute timestamp, or robustly calculate a time difference [20].

The *RADclock* is then naturally suited for the dependent clock paradigm and can be implemented in Xen as a simple read/write stateless operation using the *XenStore*, a file system that can be used as an inter-OS communication channel. After processing synchronization information received from its time server, the *RADclock* running on Dom0 writes its new clock parameters to the XenStore. On DomU, a process reads the updated clock parameters upon request and serves them to any application that needs to timestamp events. The application timestamps the event(s) of interest. These raw timestamps can then be easily converted either into a wallclock time or a time difference measured in seconds (this can even be done later off-line).

Unlike with *ntpd* and its coupled relationship to the (non-trivial) incumbent system clock code, no adjustment is passed to another dynamic mechanism, which ensures that only a single clock, clearly defined in a single module, provides universal time across Dom0 and all DomU guests.

With the above architecture, there is only one way in which a guest clock can not be strictly identical with the central Dom0 clock. The read/write operation on the XenStore is not instantaneous and it is possible that the update of clock parameters, which is slightly delayed after the processing of a new synchronization input to the *RADclock*, will result in different parameters being used to timestamp some event. In other words, the time across OSs may appear different for a short time if a timestamping function in a DomU converts a raw timestamp with outdated data. However, this is a minor issue since clock parameters change slowly, and using out of date values has the same impact as the synchronization input simply being lost, to which the clock is already robust.

In Figure 10 we measured the time required to write to the XenStore using the *RADclock* difference clock which has an accuracy well below 1 µs [20]. We present results obtained on 2 host machines with slightly different hardware architectures, namely **kultarr** (2.13 GHz Intel Core 2 Duo) and **tastiger** (3.40 GHz Intel Pentium D), that

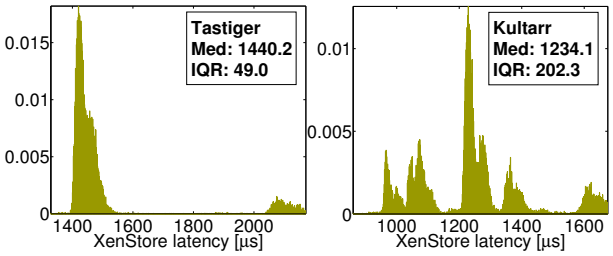


Figure 10: Distribution of clock update latency through the xenstore, **tastiger** (left, Pentium D, 3.4GHz) and **kultarr** (right, Core 2 Duo, 2.13GHz).

show respective median delays of 1.2 and 1.4 ms. Assuming a 16 s poll period, this corresponds to 1 chance out of 11,500 that the clocks would (potentially) disagree if read at some random time.

The dependent *RADclock* is ideally suited for time keeping on Xen DomU. It is a simple, stateless, standard read/write operation that is robust as it avoids the dangerous dynamics of feedback approaches, ensures that the clocks of all guests agree, and is robust to system load and power management effects. As a dependent clock solution, it saves both host and network resources and is inherently scalable. Thanks to a simple timestamping function it provides the same level of final timekeeping accuracy to all OSs.

6 A Migration-Friendly Architecture

Seamlessly migrating a running system from one physical machine to another is a key innovation of virtualization [13, 7]. However this operation becomes far from seamless with respect to timing when using *ntpd*. As mentioned in Section 3, *ntpd*'s design requires each DomU to run its own instance of the *ntpd* daemon, which is fundamentally unsuited to migration, as we now explain.

The synchronization algorithm embodied in the *ntpd* daemon is stateful. In particular it maintains a time varying estimate of the Xen Clocksource's rate-of-drift and current clock error, which in turn is defined by the characteristics of the oscillator driving the platform counter. After migration, the characteristics seen by *ntpd* change dramatically since no two oscillators drift in the same way. Although the Xen Clocksource counters on each machine nominally share the same frequency (1GHz), in practice this is only true very approximately. The temperature environment of the machine DomU migrates to can be very different from the previous one which can have a large impact, but even worse, the platform timer may be of a different nature, HPET originally and ACPI after migration for example. Furthermore, *ntpd* will also inevitably suffer from an inability to account for the time during which DomU has been halted during the migration. When DomU restarts, the reference wallclock time and last Xen Clocksource value maintained by its system clock will be quite inconsistent with the new ones, leading to extreme oscillator rate estimates. In summary, the sudden change in status of *ntpd*'s state information, from valid to almost arbitrary, will, at best, deliver a huge error immediately after migration, which we expect to decay only slowly according to *ntpd*'s usual slow convergence. At worst, the 'shock' of migration may push *ntpd* into an unstable regime from which it may never recover.

In contrast, by decomposing the time information into raw timestamps and clock parameters, as described in Section 5, the *RADclock* allows the daemon running on DomU to be stateless within an efficient dependent clock strategy. The migration then becomes trivial from a timekeeping point of view. Once migrated, DomU timestamps events of interests with its chosen counter and retrieves the *RADclock* clock parameters maintained by the new Dom0 to convert them into absolute time. DomU immediately benefits from the accuracy of the dedicated *RADclock* running on Dom0 – the convergence time is effectively zero.

The plots in Figure 11 confirm the claims above and illustrate a number of important points. In this experiment, each of **tastiger** and **kultarr** run an independent *RADclock* in Dom0. The clock error for these is remarkably similar, with an IQR below 10 µs as seen in the top plot (measured using the DAG external comparison). Here for clarity the error time series for the two Dom0 clocks have been corrected for asymmetry error, thereby allowing their almost zero inherent median error, and almost identical behavior (the air-conditioning generated oscillations overlay almost perfectly), to be clearly seen.

For the migration experiment, a single DomU OS is started on **tastiger**, and two clocks launched on it: a dependent *RADclock*, and an independent *ntpd* clock. A few hours of warm up are then given (not shown) to allow *ntpd* to fully converge. The experiment proper then begins. At the 30 minute mark DomU is migrated to **kultarr**, it migrates back to **tastiger** after 2 hours then back again after another 2, followed by further migrations with a smaller period of 30 minutes.

The resulting errors of the two migrating DomU clocks are shown in the top plot, and in a zoomed out version in the middle plot, as measured using the external comparison. Before the results, a methodological point. The dependent *RADclock* running on DomU is by construction identical to the *RADclock* running on Dom0, and so the two time series (if asymmetry corrected) would superimpose almost perfectly, with small differences owing to the different errors in the times-

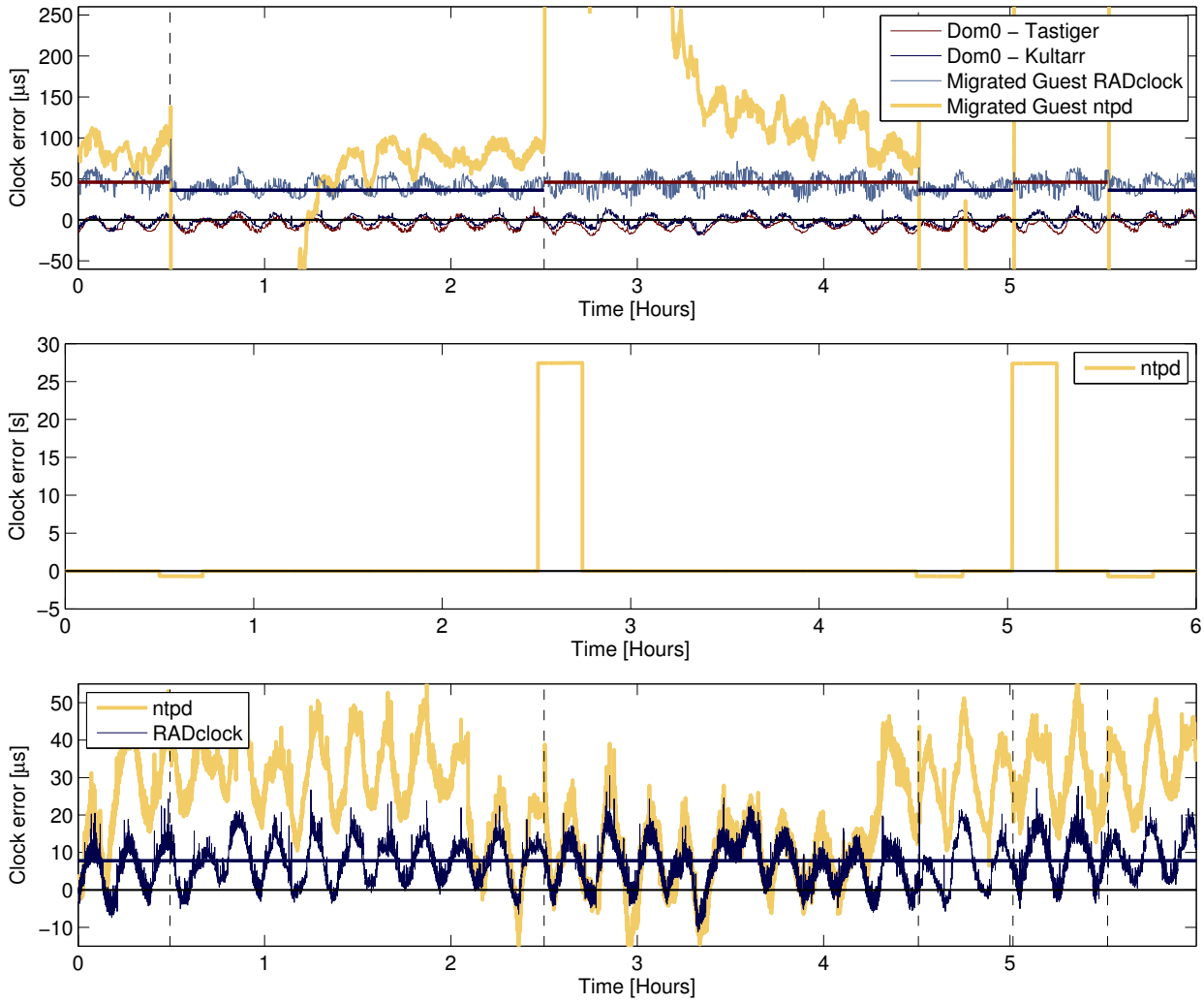


Figure 11: Clock errors under migration. Top: asymmetry corrected unmigrated *RADclock* Dom0 clocks, and (uncorrected) migrated clocks on DomU; Middle: zoom out on top plot revealing the huge size of the migration 'shock' on *ntpd*; Bottom: effect of migration load on Dom0 clocks on **kultarr**.

tamping of the separate UDP packet streams. We choose however, in the interests of fairness and simplicity of comparison, not to apply the asymmetry correction in this case, since it is not possible to apply an analogous correction to the *ntpd* error time series. As a substitute, we instead draw horizontal lines over the migrating *RADclock* time series representing the correction which *would* have been applied. No such lines can be drawn in the *ntpd* case.

Now to the results. As expected, and from the very first migration, *ntpd* exhibits extremely large errors (from -1 to 27 s!) for periods exceeding 15 minutes (see zoom in middle plot) and needs at least another hour to converge to a reasonable error level. The dependent *RADclock* on the other hand shows seamless performance with respect to the horizontal lines representing the expected jumps due to asymmetry changes as just described. These jumps are in any case small, of the order of a few microseconds. Note that these corrections are a function both of RTThost and asymmetry that are both different between **tastiger** and **kultarr**.

Finally, we present a load test comparison. The bottom plot in Figure 11 compares in detail the performance of the independent *RADclock* running on Dom0 on **kultarr**, and an independent *ntpd* clock, also running on Dom0 during the experiment (not shown previously). Whereas the *RADclock* is barely affected by the changes in network traffic and system load associated with the migrations of the DomU guest, *ntpd* shows significant deviation. In summary, not only is *ntpd* in an independent

clock paradigm incompatible with clock migration, it is also, regardless of paradigm, affected by migration occurring around it.

One could also consider the performance of an independent *RADclock* paradigm under migration. However, we expect that the associated 'migration shock' would be severe as the *RADclock* is not designed to accommodate radical changes in the underlying counter. Since the dependent solution is clearly superior from this and many other points of view, we do not present results for the independent case under migration.

7 Conclusion

Virtualization of operating systems and accurate computer based timing are two areas set to increase in importance in the future. Using Xen para-virtualization as a concrete framework, we highlighted the weaknesses of the existing timing solution, which uses independent *ntpd* synchronization algorithms (coupled to stateful software clock code) for each guest operating system. In particular, we showed that this solution is fundamentally unsuitable for the important problem of live VM migration, using both arguments founded on the design of *ntpd*, as well as detailed experiments in a hardware-validated testbed.

We reviewed the architecture of the *RADclock* algorithm, in particular its underlying feed-forward basis, the clean separation between its timestamping and synchronization aspects, and its high robustness to network and system noise (latency variability). We argued that these features make it ideal as a dependent clock solution, particularly since the clock is already set up to be read through combining a raw hardware counter timestamp with clock parameters sourced from a central algorithm which owns all the synchronization intelligence, via a commonly accessible data structure. We supported our claims by detailed experiments and side-by-side comparisons with the status quo. For the same reasons, the *RADclock* approach enables seamless and simple migration, which we also demonstrated in benchmarked experiments. The enabling of a dependent clock approach entails considerable scalability advantages and suggests further improvements through optimizing the timestamping performance of the central clock in Dom0.

As part of an examination of timestamping and counter suitability for timekeeping in general and the feed-forward paradigm in particular, we provided a detailed evaluation of the latency and accuracy of the Xen Clocksource counter, and compared it to HPET. We concluded that it works well as intended, however note that it is a complex solution created to solve a problem which will soon disappear as reliable TSC counters again become ubiquitous. The *RADclock* is suitable for use with any counter satisfying basic properties, and we showed its performance using HPET or Xen Clocksource was indistinguishable.

The *RADclock* [14] packages for Linux now support a streamlined version of the architecture for Xen described here using Xen Clocksource as the hardware counter. With the special code allowing system instrumentation and HPET access removed, no modifications to the hypervisor are finally required.

8 Acknowledgments

The *RADclock* project is partially supported under Australian Research Council's Discovery Projects funding scheme (project number DP0985673) and a Google Research Award.

We thank the anonymous reviewers and our shepherd for their valuable feedback.

9 Availability

RADclock packages for Linux and FreeBSD, software and papers, can be found at http://www.cubinlab.ee.unimelb.edu.au/radclock/.

References

[1] Xen.org History. `http://www.xen.org/community/xenhistory.html`.

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM, pp. 164–177.

[3] BROOMHEAD, T., RIDOUX, J., AND VEITCH, D. Counter Availability and Characteristics for Feed-forward Based Synchronization. In *Int. IEEE Symp. Precision Clock Synchronization for Measurement, Control and Communication (ISPCS'09)* (Brescia, Italy, Oct. 12-16 2009), IEEE Piscataway, pp. 29–34.

[4] ENDACE. Endace Measurement Systems. DAG series PCI and PCI-X cards. http://www.endace.com/networkMCards.htm.

[5] INTEL CORPORATION. IA-PC HPET (High Precision Event Timers) Specification (revision 1.0a). `http://www.intel.com/hardwaredesign/hpetspec_1.pdf`, Oct. 2004.

[6] KAMP, P. H. Timecounters: Efficient and precise timekeeping in SMP kernels. In *Proceedings of the BSDCon Europe 2002* (Amsterdam, The Netherlands, 15-17 November 2002).

[7] KEIR, C. C., CLARK, C., FRASER, K., H, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live Migration of Virtual Machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2005), pp. 273–286.

[8] KIDD, T. Intel Software Network Blogs. `http://software.intel.com/en-us/blogs/author/taylor-kidd/`.

[9] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND ZWAENEPOEL, W. Diagnosing performance overheads in the xen virtual machine environment. In *VEE '05:*

Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments (New York, NY, USA, 2005), ACM, pp. 13–23.

[10] MICROSOFT CORPORATION. Guidelines For Providing Multimedia Timer Support. Tech. rep., Microsoft Corporation, Sep. 2002. `http://www.microsoft.com/whdc/system/sysinternals/mm-timer.mspx`.

[11] MILLS, D. L. *Computer Network Time Synchronization: The Network Time Protocol*. CRC Press, Inc., Boca Raton, FL, USA, 2006.

[12] MOGUL, J., MILLS, D., BRITTENSON, J., STONE, J., AND WINDL, U. Pulse-Per-Second API for UNIX-like Operating Systems, Version 1.0. Tech. rep., IETF, 2000.

[13] NELSON, M., HONG LIM, B., AND HUTCHINS, G. Fast transparent migration for virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (2005), USENIX Association.

[14] RIDOUX, J., AND VEITCH, D. RADclock Project webpage.

[15] RIDOUX, J., AND VEITCH, D. A Methodology for Clock Benchmarking. In *Tridentcom* (Orlando, FL, USA, May 21-23 2007), IEEE Comp. Soc.

[16] RIDOUX, J., AND VEITCH, D. The Cost of Variability. In *Int. IEEE Symp. Precision Clock Synchronization for Measurement, Control and Communication (ISPCS'08)* (Ann Arbor, Michigan, USA, Sep. 24-26 2008), pp. 29–32.

[17] RIDOUX, J., AND VEITCH, D. Ten Microseconds Over LAN, for Free (Extended). *IEEE Trans. Instrumentation and Measurement (TIM) 58*, 6 (June 2009), 1841–1848.

[18] RIDOUX, J., AND VEITCH, D. Principles of Robust Timing Over the Internet. *ACM Queue, Communications of the ACM 53*, 5 (May 2010), 54–61.

[19] THE XEN TEAM. Xen Documentation. `http://www.xen.org/files/xen_interface.pdf`.

[20] VEITCH, D., RIDOUX, J., AND KORADA, S. B. Robust Synchronization of Absolute and Difference Clocks over Networks. *IEEE/ACM Transactions on Networking 17*, 2 (April 2009), 417–430.

[21] VMWARE. Timekeeping in VMware Virtual Machines. Tech. rep., VMware, May 2010. `http://www.vmware.com/files/pdf/Timekeeping-In-VirtualMachines.pdf`.