Cheap and Large CAMs for High Performance Data-Intensive Networked Systems

Ashok Anand[⋆]*, Chitra Muthukrishnan*[⋆]*, Steven Kappes*[⋆]*, Aditya Akella*[⋆] *and Suman Nath*[†]
[⋆]*UW-Madison,* [†]*Microsoft Research*

Abstract

We show how to build cheap and large CAMs, or *CLAMs*, using a combination of DRAM and flash memory. These are targeted at emerging data-intensive networked systems that require massive hash tables running into a hundred GB or more, with items being inserted, updated and looked up at a rapid rate. For such systems, using DRAM to maintain hash tables is quite expensive, while on-disk approaches are too slow. In contrast, CLAMs cost nearly the same as using existing on-disk approaches but offer orders of magnitude better performance. Our design leverages an efficient flash-oriented data-structure called BufferHash that significantly lowers the amortized cost of random hash insertions and updates on flash. BufferHash also supports flexible CLAM eviction policies. We prototype CLAMs using SSDs from two different vendors. We find that they can offer average insert and lookup latencies of 0.006ms and 0.06ms (for a 40% lookup success rate), respectively. We show that using our CLAM prototype significantly improves the speed and effectiveness of WAN optimizers.

1 Introduction

In recent years, a number of data-intensive networked systems have emerged where there is a need to maintain hash tables as large as tens to a few hundred gigabytes in size. Consider WAN optimizers [1, 2, 7, 8], for example, that maintain "data fingerprints" to aid in identifying and eliminating duplicate content. The fingerprints are 32-64b hashes computed over ∼4-8KB chunks of content. The net size of content is ∼10TB stored on disk [9]. Thus the hash table storing the mapping from fingerprints to on-disk addresses of data chunks could be ≥32GB. Just storing the fingerprints requires ∼16GB.

The hash tables in these content-based systems are also inserted into, looked up and updated frequently. For instance, a WAN optimizer connected to a 0.5Gbps link may require roughly 10,000 fingerprint lookups, insertions and updates each per second. Other examples of systems that employ similar large hash tables include data deduplication systems [4, 45], online backup services [5], and directory services in data-oriented network architectures [32, 37, 42]. These systems are becoming increasingly popular and being widely adopted [3].

This paper arises from the quest to design effective hash tables in these systems. The key requirement is that the mechanisms used be *cost-effective* for the functionality they support. That is, the mechanisms should offer a high number of hash table operations ($> 10K$) per second while keeping the overall cost low. We refer to mechanisms that satisfy these requirements as *CLAMs*, for cheap and large CAMs.

There are two possible approaches today for supporting the aforementioned systems. The first is to maintain hash tables in DRAM which can offer very low latencies for hash table operations. However, provisioning large amounts of DRAM is expensive. For instance, a 128GB RamSan DRAM-SSD offers 300K random IOs per second, but, it has a very high cost of ownership, including the device cost of $120K and an energy footprint of 650W [20]. In other words, it can support fewer than 2.5 *hash table operations per second per dollar*.

A much cheaper alternative is to store the hash tables on magnetic disks using database indexes, such as Berkeley-DB (or BDB) [6]. However, poor throughput of random inserts, lookups, and updates in BDB can severely undermine the effectiveness of the aforementioned systems and force them to run at low speeds. For example, a BDB-based WAN optimizer is effective for link speeds of only up to 10Mbps (§8). Note that existing fast stream databases [22, 11, 14] and wire-speed data collection systems [24, 29] are not suitable as CLAMs as they do not include any archiving and indexing schemes.

In this paper we design and evaluate an approach that is *1-2 orders of magnitude better* in terms of hash operations/sec/$ compared to *both* disk-based and DRAM-based approaches. Our approach uses a commodity two-level storage/memory hierarchy consisting of some DRAM and a much larger amount of *flash storage* (could be flash memory chips or solid state disks (SSDs)). Our design consumes most of the I/Os in the DRAM, giving low latency and high throughput I/Os compared to a flash-only design. On the other hand, using flash allows us to support a large hash table in a cheaper way than DRAM-only solutions. We choose flash over magnetic disks for its many superior properties, such as, higher I/O per second per dollar and greater reliability, as well as far superior power efficiency compared to both DRAM and magnetic disks. Newer generation of SSDs are rapidly getting bigger and cheaper. Configuring our design with 4GB of memory and 80GB of flash, for instance, costs as little as $400 using current hardware.

Despite flash's attractive I/O properties, building a CLAM using flash is challenging. In particular, since the available DRAM is limited, a large part of the hash

table must be stored in flash (unlike recent works, e.g., FAWN [13], where the hash index is fully in DRAM). Thus, hash insertion would require random I/Os, which are expensive on flash. Moreover, the granularity of a flash I/O is orders of magnitude bigger than that of an individual hash table operation in the systems we target. Thus, unless designed carefully, the CLAM could perform poorer than a traditional disk-based approach.

To address these challenges, we introduce a novel data structure, called BufferHash. BufferHash represents a careful synthesis of prior ideas along with a few novel algorithms. A key idea behind BufferHash is that instead of performing individual random insertions directly on flash, DRAM can be used to buffer multiple insertions and writes to flash can happen in a *batch*. This shares the cost of a flash I/O operation across multiple hash table operations, resulting in a better amortized cost per operation. Like a log-structured file system [39], batches are written to flash sequentially, the most efficient write pattern for flash. The idea of batching operations to amortize I/O costs has been used before in many systems [15, 28]. However using it for hash tables is novel, and it poses several challenges for flash storage.

Fast lookup: With batched writes, a given $(key, value)$ pair may reside in any prior batch, depending on the time it was written out to flash. A naive lookup algorithm would examine all batches for the key, which would incur high and potentially unacceptable flash I/O costs. To reduce the overhead of examining on-flash batches, BufferHash (i) partitions the key space to limit the lookup to one partition, instead of the entire flash (similar to how FAWN spreads lookups across multiple "wimpy nodes") [13]), and (ii) uses in-memory Bloom filters (as Hyperion [23] does) to efficiently determine a small set of batches that may contain the key.

Limited flash: In many of the streaming applications mentioned earlier, insertion of new $(key, value)$ entries into the CLAM requires creating space by evicting old keys. BufferHash uses a novel age-based internal organization that naturally supports bulk evictions of old entries in an I/O-efficient manner. BufferHash also supports other flexible eviction policies (e.g. priority-based removal) to match different application needs, albeit at additional performance cost. Existing proposals for indexing archived streaming data ignore eviction entirely.

Updates: Since flash does not support efficient update or deletion, modifying existing $(key, value)$ mappings *in situ* is expensive. To support good update latencies, we adopt a *lazy update* approach where *all* value mappings, including deleted or updated ones, are temporarily left on flash and later deleted in batch during eviction. Such lazy updates have been previously used in other contexts, such as buffer-trees [15] and lazy garbage collection in log-structured file systems [39].

Performance tuning: The unique I/O properties of flash demand careful choice of various parameters in our design of CLAMs, such as the amount of DRAM to use, and the sizes of batches and Bloom filters. Suboptimal choice of these parameters may result in poor overall CLAM performance. We model the impact of these parameters on latencies of different hash table operations and show how to select the optimal settings.

We build CLAM prototypes using SSDs from two vendors. Using extensive analysis based on a variety of workloads, we study the latencies supported in each case and compare the CLAMs against popular approaches such as using Berkeley-DB (BDB) on disk. In particular, we find that our Intel SSD-based CLAM offers an average *insert* latency of 0.006ms compared to 7ms from using BDB on disk. For a workload with 40% hit rate, the average *lookup* latency is 0.06ms for this CLAM, but 7ms for BDB. Thus, our CLAM design can yield 42 lookups/sec/$ and 420 insertions/sec/$ which is 1-2 orders of magnitude better than RamSan DRAM-SSD (2.5 hash operations/sec/$). The superior energy efficiency of flash and rapidly declining prices compared to DRAM [21] mean that the gap between our CLAM design and DRAM-based solutions is greater than indicated in our evaluation and likely to widen further. Finally, using real traces, we study the benefits of employing the CLAM prototypes in WAN optimizers. Using a CLAM, the speed of a WAN optimizer can be improved ≥ 10X compared to using BDB (a common choice today [2]).

Our CLAM design marks a key step in building fast and effective indexing support for high-performance content-based networked systems. We do not claim that our design is final. We speculate that there may be smarter data structures and algorithms, that perhaps leverage newer memory technologies (e.g. Phase Change Memory), offering much higher hash operations/sec/$.

2 Related Work

In this section, we describe prior work on designing data structures for flash and recent proposals for supporting data-intensive streaming networked systems.

Data structures for flash: Recent work has shown how to design efficient data structures on flash memory. Examples include MicroHash [44], a hash table and FlashDB [36], a B-Tree index. Unlike BufferHash, these data structures are designed for memory-constrained embedded devices where the design goal is to optimize energy usage and minimize memory footprint—latency is typically not a design goal. For example, a lookup operation in MicroHash may need to follow multiple pointers to locate the desired key in a chain of flash blocks and can be very slow. Other recent works on designing efficient codes for flash memory to increase its effective capacity [30, 27] are orthogonal to our work, and BufferHash

can be implemented on top of these codes.

A flash-based key-value store: Closely related to our design of CLAMs is the recent FAWN proposal [13]. FAWN-KV is a clustered key-value storage built on a large number of tiny nodes that each use embedded processors and small amounts of flash memory. There are crucial differences between our CLAM design and FAWN. First, FAWN assumes that each wimpy node can keep its hash index in DRAM. In contrast, our design targets situations where the actual hash index is bigger than available DRAM and hence part of the index needs to be stored in flash. In this sense, our design is complementary to FAWN; if the hash index in each wimpy node gets bigger than its DRAM, it can use BufferHash to organize the index. Second, being a cluster-based solution, FAWN optimizes for throughput, not for latency. As the evaluation of FAWN shows, some of the lookups can be very slow ($> 500ms$). In contrast, our design provides better worst-case latency ($< 1ms$), which is crucial for systems such as WAN optimizers. Finally, FAWN-KV does not focus on efficient eviction of indexed data.

Along similar lines is HashCache [16], a cache that can run on cheap commodity laptops. It uses an in-memory index for objects stored on disk. Our approach is complementary to HashCache, just as it is with FAWN.

DRAM-only solutions: DRAM-SSDs provide extremely fast I/Os, at the cost of high device cost and energy footprint. For example, a 128GB RamSan device can support 300K IOPS, but costs $120K\$$ and consumes 650W [20]. A cheaper alternative from Violin memory supports 200K IOPS, but still costs around $50K\$$ [40]. Our CLAM prototypes significantly outperform traditional hash tables designed in these DRAM-SSDs in terms of operations/s/$.

Large scale streaming systems: Hyperion [23] enables archival, indexing, and on-line retrieval of high-volume data streams. However, Hyperion does not suit the applications we target as it does not offer CAM-like functionality. For example, to lookup a key, Hyperion may need to examine prohibitively high volume of data resulting in a high latency. Second, it does not consider using flash storage, and hence does not aim to optimize design parameters for flash. Finally, it does not support efficient update or eviction of indexed data.

Existing data stream systems [11, 14, 22] do not support queries over archived data. StreamBase [41] supports archiving data and processing queries over past data; but the data is archived in conventional hash or B-Tree-indexed tables, both of which are slow and are suitable only for offline queries. Endace DAG [24] and CoMo [29] are designed for wire-speed data collection and archiving; but they provide no query interface. Existing DBMSs can support CAM-like functionalities. However, they are designed neither for high update and lookup rates (see [14]) nor for flash storage (see [36]).

3 Motivating Applications

In this section, we describe three networked systems that could benefit from effective mechanisms for building and maintaining large hash tables that can be written to and looked up at a very fast rate.

WAN optimization. WAN optimizers [1, 2, 8, 7] are used by enterprises and data centers to improve network utilization by looking for and suppressing redundant information in network transfers. A WAN optimizer computes fingerprints of each arriving data object and looks them up in a hash table of fingerprints found in prior content. The fingerprints are 32-64b hashes computed over $\sim$4-8KB data chunks. Upon finding a match, the corresponding duplicate content is removed, and the "compressed object" is transmitted to the destination, where it gets reconstructed. Fingerprints for the original object are inserted into the index to aid in future matches. The content is typically $\geq$10TB in net size [10]. Thus the fingerprint hash tables could be $\geq$32GB.

Consider a WAN optimizer connected to a heavily-loaded 500Mbps link. Roughly 10,000 content fingerprints are created per second. Depending on the implementation, three scenarios may arise during hash insertion and lookup: (1) lookups for upcoming objects are held-up until prior inserts complete, or (2) upcoming objects are transmitted without fingerprinting and lookup, or (3) insertions are aborted mid-way and upcoming objects looked up against an "incomplete index." Fast support for insertions and lookups can improve all three situations and help identify more content redundancy. In §8, we show that a BDB-based WAN optimizer can function effectively only at low speeds (10Mbps) due to BDB's poor support for random insertions and lookups, even if BDB is maintained on an Intel SSD. A CLAM-based WAN optimizer using a low-end transcend SSD that is an order of magnitude slower than an Intel SSD is highly effective even at 200-300Mbps.

Data deduplication and backup. Data deduplication [4] is the process of suppressing duplicate content from enterprise data leaving only one copy of the data to be stored for archival. Prior work suggests that data sets in de-dup systems could be roughly 8-10TB and employ 20GB indexes [4, 45].

A time-consuming activity in deduplication is merging data sets and the corresponding indexes. To merge a smaller index into a larger one, fingerprints from the latter dataset need to be looked up, and the larger index updated with any new information. We estimate that merging fingerprints into a larger index using Berkeley-DB could take as long as 2hrs. In contrast, our CLAM prototypes can help the merge finish in under 2mins. We note that a similar set of challenges arise in online backup services [5] which allow users to constantly, and in an

online fashion, update a central repository with "diffs" of the files they are editing, and to retrieve changes from any remote location on demand.

Central directory for a data-oriented network. Recent proposals argue for a new resolution infrastructure to dereference content names directly to host locations [32, 37, 42]. The names are hashes computed over chunks of content inside data objects. As new sources of data arise or as old sources leave the network, the resolution infrastructure should be updated accordingly. To support scalability, the architectures have conventionally relied on a distributed resolution mechanism based on DHTs [32, 37, 42]. However, in some deployment scenarios (e.g. a large corporation), the resolution may have to be provided by a trusted central entity. To ensure high availability and throughput for a large user-base, the centralized deployment should support fast inserts and efficient lookups of the mappings. The CLAMs we design can support such an architecture effectively.

4 Flash Storage and Hash Tables

Flash provides a non-volatile memory store with several significant benefits over typical magnetic hard disks such as fast random reads ($\ll$ 1 ms), power-efficient I/Os ($<$1 Watt), better shock resistance, etc. [33, 36]. However, because of the unique characteristics of flash storage, applications designed for flash should follow a few well-known design principles: **(P1)** Applications should avoid random writes, in-place updates, and sub-block deletions as they are significantly expensive on flash. For example, updating a single 2KB page in-place requires first erasing an entire erase block (128KB-256KB) of pages, and then writing the modified block in its entirety. As shown in [35], such operations are over two orders of magnitude slower than sequential writes, out-of-place updates, and block deletions respectively, on both flash chips and SSDs. **(P2)** Since reads and writes happen at the granularity of a flash page (or an SSD sector), an I/O of size smaller than a flash page (2KB) costs at least as much as a full-page I/O. Thus, applications should avoid small I/Os if possible. **(P3)** The high fixed initialization cost of an I/O can be amortized with a large I/O size [12]. Thus, applications should batch I/Os whenever possible. In designing flash-based CLAMs using BufferHash, we follow these design principles.

A conventional hash table on flash. Before going into the details of our BufferHash design, it might be useful to see why a conventional hash table on flash is likely to suffer from poor performance. Successive keys inserted into a hash table are likely to hash to random locations in the hash table; therefore, values written to those hashed locations will result in random writes, violating the design principle **P1** above.

Updates and deletions are immediately applied to a conventional hash table, resulting in in-place updates and sub-block deletions (since each hashed value is typically much smaller than a flash block), and violation of **P1**.

Since each hashed value is much smaller than a flash page (or an SSD sector), inserting a single key in an in-flash hash table violates principles **P2** and **P3**. Violation of these principles results in a poor performance of a conventional hash table on flash, as we demonstrate in §7.

One can try to improve the performance by buffering a part of the hash table in DRAM and keeping the remaining in flash. However, since hash operations exhibit negligible locality, such a flat partitioning has very little performance improvement. Recent research has confirmed that a memory buffer is practically useless for external hashing for a read-write mixed workload [43].

5 The BufferHash Data Structure

BufferHash is a flash-friendly data structure that supports hash table-like operations on $(key, value)$ pairs[1]. The key idea underlying BufferHash is that instead of performing individual insertions/deletions one at a time to the hash table on flash, we can perform multiple operations all at once. This way, the cost of a flash I/O operation can be shared among multiple insertions, resulting in a better amortized cost for each operation (similar to buffer trees [15] and group commits in DBMS and file systems [28]). For simplicity, we consider only insertion and lookup operations for now; we will discuss updates and deletions later.

To allow multiple insertions to be performed all at once, BufferHash operates in a lazy batched manner: it accumulates insertions in small in-memory hash tables (called *buffers*), without actually performing the insertions on flash. When a buffer fills up, all inserted items are pushed into flash in a batch. For I/O efficiency, items pushed from a buffer to flash are sequentially written as a new hash table, instead of performing expensive update to existing in-flash hash tables. Thus, at any point of time, the flash contains a large number of small hash tables. During lookup, a set of Bloom filters is used to determine which in-flash hash tables may contain the desired key, and only those tables are retrieved from flash. At a high level, the efficiency of this organization comes from batch I/O and sequential writes during insertions. Successful lookup operations may still need random page reads, however, random page reads are almost as efficient as sequential page reads in flash.

5.1 A Super Table

BufferHash consists of multiple *super tables*. Each super table has three main components: a buffer, an incarnation table, and a set of Bloom filters. These components are

[1]For clarity purposes we note that BufferHash is a data-structure while a CLAM is BufferHash applied atop DRAM and flash.

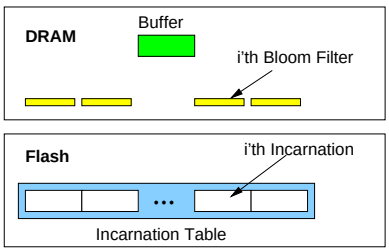


Figure 1: *A Super Table*

organized in two levels of hierarchy, as shown in Figure 1. Components in the higher level are maintained in DRAM, while those in the lower level are maintained in flash.

Buffer. This is an in-memory hash table where all newly inserted hash values are stored. The hash table can be built using existing fast algorithms such as multiple-choice hashing [18, 31]. A buffer can hold a fixed maximum number of items, determined by its size and the desired upper bound of hash collisions. When the number of items in the buffer reaches its capacity, the entire buffer is flushed to flash, after which the buffer is re-initialized for inserting new keys. The buffers flushed to flash are called *incarnations*.

Incarnation table. This is an in-flash table that contains old and flushed incarnations of the in-memory buffer. The table contains k incarnations, where k denotes the ratio of the size of the incarnation table and the buffer. The table is organized as a circular list, where a new incarnation is sequentially written at the list-head. To make space for a new incarnation, the oldest incarnation, at the tail of the circular list, is evicted from the table.

Depending on application's eviction policy, some items in an evicted incarnation may need to be retained and are re-inserted into the buffer (details in §5.1.2).

Bloom filters. Since the incarnation table contains a sequence of incarnations, the value for a given hash key may reside in any of the incarnations depending on its insertion time. A naive lookup algorithm for an item would examine all incarnations, which would require reading all incarnations from flash. To avoid this excessive I/O cost, a super table maintains a set of in-memory Bloom filters [19], one per incarnation. The Bloom filter for an incarnation is a compact signature built on the hash keys in that incarnation. To search for a particular hash key, we first test the Bloom filters for all incarnations; if any Bloom filter matches, then the corresponding incarnation is retrieved from flash and looked up for the desired key. Bloom filter-based lookups may result in false positive; thus, a match could be indicated even though there is none, resulting in unnecessary flash I/O. As the filter size increases, the false positive rate drops, resulting in lower I/O overhead. However, since the available DRAM is limited, filters cannot be too large in size. We examine the tradeoff in §6.4.

The Bloom filters are maintained as follows: When a buffer is initialized after a flush, a Bloom filter is created for it. When items are inserted into the buffer, the Bloom filter is updated with the corresponding key. When the buffer is flushed as an incarnation, the Bloom filter is saved in memory as the Bloom filter for that incarnation. Finally, when an incarnation is evicted, it's Bloom filter is discarded from memory.

5.1.1 Super Table Operations

A super table supports all standard hash table operations.

Insert. To insert a $(key, value)$ pair, the value is inserted in the hash table in the buffer. If the buffer does not have space to accommodate the key, the buffer is flushed and written as a new incarnation in the incarnation table. The incarnation table may need to evict an old incarnation to make space.

Lookup. A key is first looked up in the buffer. If found, the corresponding value is returned. Otherwise, in-flash incarnations are examined in the order of their age until the key is found. To examine an incarnation, first its Bloom filter is checked to see if the incarnation might include the key. If the Bloom filter matches, the incarnation is read from flash, and checked if it really contains the key. Note that since each incarnation is in fact a hash table, to lookup a key in an incarnation, only the relevant part of the incarnation (e.g., a flash page) can be read directly.

Update/Delete. As mentioned earlier, flash does not support small updates/deletions efficiently; hence, we support them in a lazy manner. Suppose a super table contains an item (k, v), and later, the item needs to be updated with the item (k, v'). In a traditional hash table, the item (k, v) is immediately replaced with (k, v'). If (k, v) is still in the buffer when (k, v') is inserted, we do the same. However, if (k, v) has already been written to flash, replacing (k, v) will be expensive. Hence, we simply insert (k, v') without doing anything to (k, v). Since the incarnations are examined in order of their age during lookup, if the same key is inserted with multiple updated values, the latest value (in this example, v') is returned by a lookup. Similarly, for deleting a key k, a super table does not delete the corresponding item unless it is still in the buffer; rather the deleted key is kept in a separate list (or, a small in-memory hash table), which is consulted before lookup—if the key is in the delete list, it is assumed to be deleted even though it is present in some incarnation. Lazy update wastes space on flash, as outdated items are left on flash; the space is reclaimed during incarnation eviction.

5.1.2 Incarnation Eviction

In a streaming application, BufferHash may have to evict old in-flash items to make space for new items. The de-

cision of what to evict depends on application policy.

For I/O efficiency, BufferHash evicts items in granularity of an incarnation. Since each incarnation is an independent hash table, discarding a part of it may require expensive reorganization of the table and expensive I/O to write it back to flash. To this end, BufferHash provides two basic eviction primitives. The *full discard* primitive entirely evicts the oldest incarnation. The *partial discard* primitive also evicts the oldest incarnation, but it scans through all the items in the incarnation before eviction, selects some items to be retained (based on a specified policy), and re-inserts them into the buffer. Given these two basic primitives, applications can configure BufferHash to implement different eviction policies as follows.

FIFO. The full discard primitive naturally implements the FIFO policy. Since items with similar ages (i.e., items that are flushed together from the buffer) are clustered in the same incarnation, discarding the oldest incarnation evicts the oldest items. Commercial WAN optimizers work in this fashion [8, 2].

LRU. An LRU policy can be implemented via the full discard mechanism with one additional mechanism: on every use of an item not present in the buffer, the item is re-inserted. Intuitively, a recently used item will be present in a recent incarnation, and hence it will still be present after discarding the oldest incarnation. This implementation incurs additional space overhead as the same item can be present in multiple incarnations.

Update-based eviction. With a workload with many deletes and updates, BufferHash uses the partial discard mechanism to discard items that have been deleted or updated. The former can be determined by examining the in-memory delete list, while the latter can be determined by checking the in-memory Bloom filters.

Priority-based eviction. In a priority-based policy, an item is discarded if its priority is less than a threshold (the threshold can change over time, as in [35]). It can be implemented with the partial discard primitive, where an item in the discarded incarnation is re-inserted if its current priority is above a threshold.

The FIFO policy is the most efficient, and the default policy in BufferHash. The other policies incur additional space and latency overhead due to more frequent buffer flushes and re-insertion.

Note that BufferHash may not be able to *strictly* follow an eviction policy other than FIFO if enough slow storage is not available. Suppose an item is called *live* if it is supposed to be present in the hash table under a given eviction policy (e.g., for the update-based eviction policy, the item has not been updated or deleted), and *dead* otherwise. BufferHash is supposed to evict *only* the dead items, and it does so if the flash has enough space to hold all live and unevicted dead items. On the other hand, if available flash space is limited and there are not enough dead items to evict in order to make room for newer items, BufferHash is forced to evict *live* items in a *FIFO order*. [2] We note that this sort of behavior is unavoidable in *any* storage scheme dealing with too many items to be fit in a limited amount of storage.

5.1.3 Bit-slicing with a Sliding Window

To support efficient Bloom filter lookup, we organize the Bloom filters for all incarnations within a super table in bit-sliced fashion [26]. Suppose a super table contains k incarnations, and the Bloom filter for each incarnation has m bits. We store all k Bloom filters as m k-bit slices, where the i'th slice is constructed by concatenating bit i from each of the k Bloom filters. Then, if a Bloom filter uses h hash functions, we apply them on the key x to get h bit positions in a Bloom filter, retrieve h bit slices at those positions, compute bit-wise AND of those slices. Then, the positions of 1-bits in this aggregated slice, which can be looked up from a pre-computed table, represent the incarnations that may contain the key x.

As new incarnations are added and old ones evicted, bit slices need to be updated accordingly. A naive approach would reset the left-most bits of all m bit-slices on every eviction, further increasing the cost of an eviction operation. To avoid this, we append w extra bits with every bit-slice, where w is the size of a word that can be reset to 0 with one memory operation. Within each $(k+w)$-bit-slice, a window of k bits represent the Bloom filter bits of k current incarnations, and only these bits are used during lookup. After an incarnation is evicted, the window is shifted one bit right. Since the bit falling off the window is no longer used for lookup, it can be left unchanged. When the window has shifted w bits, entire w-bit words are reset to zero at once, resulting in a small amortized cost. The window wraps around after it reaches the end of a bit-slice. For lack of space we omit the details, which can be found in [12].

5.2 Partitioned Super Tables

Maintaining a single super table is not scalable because the buffer and individual incarnations will become very large with a large available DRAM. As the entire buffer is flushed at once, the flushing operation can take a long time. Since flash I/Os are blocking operations, lookup operations that go to flash during this long flushing period will block (insertions can still happen as they go to in-memory buffer). Moreover, an entire incarnation from the incarnation table is evicted at a time, increasing the eviction cost with partial discard.

[2] With the update-based eviction policy, a live item can also be evicted if the in-memory Bloom filter incorrectly concludes that the item has been updated. However, the probability is small (equals to the false positive rate of the Bloom filters).

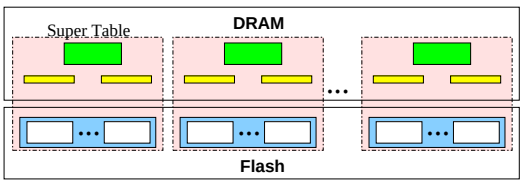


Figure 2: *A BufferHash with multiple super tables*

BufferHash avoids this problem by partitioning the hash key space and maintaining one super table for each partition (Figure 2): Suppose each hash key has $k = k_1 + k_2$ bits; then, BufferHash maintains 2^{k_1} super tables. The first k_1 bits of a key represents the index of the super table containing the key, while the last k_2 bits are used as the key within the particular super table.

Partitioning enables using small buffers in super tables, thus avoiding the problems caused by a large buffer. However, we show in §6.4 that too many partitions (i.e., very small buffers) can also adversely affect performance. We show how to choose the number of partitions for good performance. For example, we show for flash chips that the number of partitions should be such that the size of a buffer matches the flash block size.

BufferHash with multiple super tables can be implemented on a flash chip by statically partitioning it and allocating each partition to a super table. A super table writes its incarnations in its partition in a circular way—after the last block of the partition is written, the first block of the partition is erased and the corresponding incarnations are evicted. However, this approach may not be optimal for an SSD, where a Flash Translation Layer (FTL) hides the underlying flash chips. Even though writes within a single partition are sequential, writes from different super tables to different partitions may be interleaved, resulting in a performance worse than a single sequential write (see [17] for empirical results). To deal with that, BufferHash uses the entire SSD as a single circular list and writes incarnations from different super tables sequentially, in the order they are flushed to the flash. (This is in contrast to the log rotation approach of Hyperion [23] that provides FIFO semantics for each partition, instead of the entire key space.) Note that partitioning also naturally supports using multiple SSDs in parallel, by distributing partitions to different SSDs. This scheme, however, spreads the incarnations of a super table all over the SSD. To locate incarnations for a given super table, we maintain their flash addresses along with their Bloom filters and use the addresses during lookup.

6 Analysis of Costs

In this section, we first analyze the I/O costs of insertion and lookup operations in CLAMs built using BufferHash for flash-based storage, and then use the analytical results to determine optimal values of two important parameters of BufferHash. We use the notations in Table 1.

Symbol	Meaning
N	Total number of items inserted
M	Total memory size
B	Total size of buffers
b	Total size of Bloom filters
k	Number of incarnations in a super table
F	Total flash size
s	Average size taken by a hash entry
h	Number of hash functions
B'	Size of a single buffer (=B/n)
S_p	Size of a flash page/sector
S_b	Size of a flash block

Table 1: *Notations used in cost analysis.*

6.1 Insertion Cost

We now analyze the amortized and the worst case cost of an insertion operation. We assume that BufferHash is maintained on a flash chip; later we show how the results can be trivially extended to SSDs. Based on empirical results [12], we use linear cost functions for flash I/Os—reading, writing, and erasing x bits, at appropriate granularities, cost $a_r + b_r x$, $a_w + b_w x$, and $a_e + b_e x$, respectively.

Consider a workload of inserting N keys. Most insertions are consumed in buffers, and hence do not need any I/O. However, expensive flash I/O occurs when a buffer fills and is flushed to flash. Each flush operation involves three different types of I/O costs. First, each flush requires writing $n_i = \lceil B'/S_p \rceil$ pages, where B' is the size of a buffer in a super table, and S_p is the size of a flash page (or an SSD sector). This results in a write cost of

$$C_1 = a_w + b_w n_i S_p$$

Second, each flush operation requires evicting an old incarnation from the incarnation table. For simplicity, we consider full discard policy for an evicted incarnation. Note that each incarnation occupies $n_i = \lceil B'/S_p \rceil$ flash pages, and each flash block has $n_b = S_b/S_p$ pages, where S_b is the size of a flash block. If $n_i \geq n_b$, every flush will require erasing flash blocks; otherwise, only n_i/n_b fraction of the flushes will require erasing blocks. Finally, during each erase, we need to erase $\lceil n_i/n_b \rceil$ flash blocks. Putting all together, we get the erase cost of a single flush operation as

$$C_2 = \text{Min}(1, n_i/n_b)(a_e + b_e \lceil n_i/n_b \rceil S_b)$$

Finally, a flash block to be erased may contain valid pages (from other incarnations), which must be backed up before erase and copied back after erase. This can happen because flash can be erased only at the granularity of a block and an incarnation to be evicted may occupy only part of a block. In this case, $p' = (n_b - n_i) \mod n_b$ pages must be read and written during each flush. This results in a copying cost of

$$C_3 = a_r + p' b_r S_p + a_w + p' b_w S_p$$

Amortized cost. Consider insertion of N keys. If each hash entry occupies a space of s, each buffer can

hold B'/s entries, and hence buffers will be flushed to flash a total of $n_f = Ns/B'$ times. Thus, the amortized insertion cost is

$$C_{amortized} = n_f(C_1+C_2+C_3)/N = (C_1+C_2+C_3)s/B'$$

Note that the cost is independent of N and inversely proportional to the buffer size B'.

Worst case cost. An insert operation experiences the worst-case performance when the buffer for the key is full, and hence must be flushed. Thus, the worst case cost of an insert operation is

$$C_{worst} = C_1 + C_2 + C_3$$

SSD. The above analysis extends to SSDs. Since the costs C_2 and C_3 in an SSD are handled by its FTL, the overheads of erasing blocks and copying valid pages are reflected in its write cost parameters a_w and b_w. Hence, for an SSD, we can ignore the cost of C_2 and C_3. Thus, we get: $C_{amortized} = C_1 s/B'$ and $C_{worst} = C_1$.

6.2 Lookup Cost

A lookup operation in a super table involves first checking the buffer for the key, checking the Bloom filters to determine which incarnations may contain the key, and reading a flash page for each of those incarnations to actually lookup the key. Since a Bloom filter may produce false positives, some of these incarnations may not contain the key, and hence some of the I/Os may redundant.

Suppose BufferHash contains n_t super tables. Then, each super table will have $B' = B/n_t$ bits for its buffer, and $b' = b/n_t$ bits for Bloom filters. In steady state, each super table will contain $k = (F/n_t)/(B/n_t) = F/B$ incarnations. Each incarnation contains $n' = B'/s$ entries, and a Bloom filter for an incarnation will have $m' = b'/k$ bits. For a given m' and n', the false positive rate of a Bloom filter is minimized with $h = m' \ln 2/n'$ hash functions [19]. Thus, the probability that a Bloom filter will return a hit (i.e., indicating the presence of a given key) is given by $p = (1/2)^h$. For each hit, we need to read a flash page. Since there are c incarnations, the expected flash I/O cost is given by

$$\begin{aligned} C_{lookup} &= kpc_r = k(1/2)^h c_r \\ &= F/B(1/2)^{bs \ln 2/F} c_r \end{aligned}$$

where c_r is the cost of reading a single flash page from a flash chip, or a single sector from an SSD.

6.3 Discussion

The above analysis can provide insights into benefits and overheads of various BufferHash components that are not used in traditional hash tables. Consider a traditional hash table stored on an SSD; without any buffering, each insertion operation would require one random sector write. Suppose, sequentially writing a buffer of size B' is α times more expensive than randomly writing one sector of an SSD. α is typically small even for a buffer significantly bigger than a sector, mainly due to two reasons. First, sequential writes are significantly cheaper than random writes in most existing SSDs. Second, writing multiple consecutive sectors in a batch has better per sector latency. In fact, for many existing SSDs, the value of α is less than 1 even for a buffer size of $256KB$ (e.g., 0.39 and 0.36 for Samsung and MTron SSDs respectively). For Intel SSD, the gap between sequential and random writes is small; still the value of α is less than 10 due to I/O batching.

Clearly, the worst case insertion cost into a CLAM using BufferHash for flash is α times more expensive than that of a traditional hash table without buffering—a traditional hash table requires writing a random sector, while BufferHash sequentially writes the entire buffer. As discussed above, the value of α is small for existing SSDs. On the other hand, our previous analysis shows that the amortized insertion cost of BufferHash on flash is at least $\frac{B'}{\alpha s}$ times less than a traditional hash table, even if we assume random writes required by traditional hash table are as cheap as sequential writes required by BufferHash on flash. In practice, random writes are more expensive, and therefore, the amortized insertion cost when using BufferHash on flash is even more cheap than that of a traditional hash table.

Similarly, a traditional hash table on flash will need one read operation for each lookup operation, even for unsuccessful lookups. In contrast, the use of Bloom filter can significantly reduce the number of flash reads for unsuccessful lookups. More precisely, if the Bloom filters are configured to provide a false positive rate of p, use of Bloom filter can reduce the cost of an unsuccessful lookup by a factor of $1/p$. Note that the same benefit can be realized by using Bloom filters with a traditional hash table as well. Even though BufferHash maintains multiple Bloom filters over different partitions and incarnations, the total size of all Bloom filters will be the same as the size of a single Bloom filter computed over all items. This is because for a given false positive rate, the size of a Bloom filter is proportional to the number of unique items in the filter,

6.4 Parameter Tuning

Tuning BufferHash for good CLAM performance requires tuning two key parameters. First, one needs to decide how much DRAM to use, and if a large enough DRAM is available, how much of it is to allocate for buffer and how much to allocate for Bloom filters. Second, once the total size of in-memory buffers is decided, one needs to decide how many super tables to use. We use the cost analysis above to address these issues.

Optimal buffer size. Assume that the total memory size is M bits, of which B bits are allocated for (all) buffers (in all super tables) and $b = M - B$ bits are allocated for Bloom filters. Our previous analysis shows that the value of B does not directly affect insertion cost; however, it affects lookup cost. So, we would like to find the optimal value of B that minimizes the expected lookup cost.

Intuitively, the size of a buffer poses a tradeoff between the total number of incarnations and the probability of an incarnation being read from flash during lookup. As our previous analysis showed, the I/O cost is proportional to the product of the number of incarnations and the hit rate of Bloom filters. On one hand, reducing buffer size increases the number of incarnations, increasing the cost. On the other hand, increasing buffer size leaves less memory for Bloom filters, which increases its false positive rate and I/O cost.

We can use our previous analysis to find a sweet-spot. Our analysis showed that the lookup cost is given by $C = F/B \cdot (1/2)^{(M-B)s \ln 2/F} \cdot c_r$. The cost C is minimized when $dC/dB = 0$, or, equivalently $d(log_2(C))/dB = 0$. Solving this equation gives the optimal value of B as,

$$B_{opt} = \frac{F}{s(\ln 2)^2} \approx \frac{2F}{s}$$

Interestingly, this optimal value of B does not depend on M; rather, it depends only on the total size F of flash and the average space s taken by each hashed item. Thus, given some memory of size $M > B$, we should use $\approx 2F/s$ bits for buffers, and the remaining for Bloom filters. If additional memory is available, that should be used only for Bloom filters, and not for the buffers.

Total memory size. We can also determine how much total memory to use for BufferHash. Intuitively, using more memory improves lookup performance, as this allows using larger Bloom filters and lowering false positive rates. Suppose, we want to limit the I/O overhead due to false positives to C_{target}. Then, we can determine b', the required size of Bloom filters as follows.

$$\begin{aligned} C_{target} &\geq \frac{F}{B}\left(\frac{1}{2}\right)^{b's\ln 2/F} \cdot c_r \\ b' &\geq \frac{F}{s(\ln 2)^2} \ln\left(\frac{s(\ln 2)^2 c_r}{C_{target}}\right) \end{aligned}$$

Figure 3 shows required size of a Bloom filter for different expected I/O overheads. As the graph shows, the benefit of using large Bloom filter diminishes after a certain size. For example, for BufferHash with 32GB flash and 16 bytes per entry (effective size of 32 bytes per entry for 50% utilization of hashtables), allocating 1GB for all Bloom filters is sufficient to limit the expected I/O overhead C_{target} below 1ms.

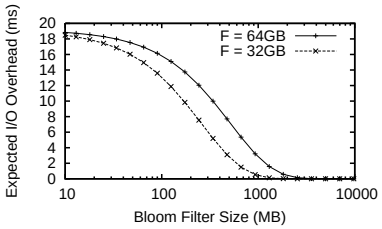


Figure 3: *Expected I/O overhead vs Bloom filter size*

Hence, in order to limit I/O overhead during lookup to C_{target}, BufferHash requires $(B_{opt} + b')$ bits of memory, of which B_{opt} is for buffers and the rest for Bloom filters.

Number of super tables. Given a fixed memory size B for all buffers, the number of super tables determines the size B' of a buffer within a super table. As our analysis shows, B' does not affect the lookup cost; rather, it affects the amortized and worst case cost of insertion. Thus, B' should be set to minimize insertion cost.

Figure 4 shows the insertion cost of using BufferHash, based on our previous analysis, on two flash-based media. (The SSD performs better because it uses multiple flash chips in parallel.). For the flash chip, both amortized and worst-case cost minimize when the buffer size B' matches the flash *block size*. The situation is slightly different for SSDs; as Figure 4(b) and (c) show, a large buffer reduces average latency but increases worst case latency. An application should use its tolerance for average- and worst-case latencies and our analytical results to determine the desired size of B' and the number of super tables B/B'.

7 Implementation and Evaluation

In this section, we measure and dissect the performance of various hash table operations in our CLAM design under a variety of workloads. Our goal is to answer the following key questions:

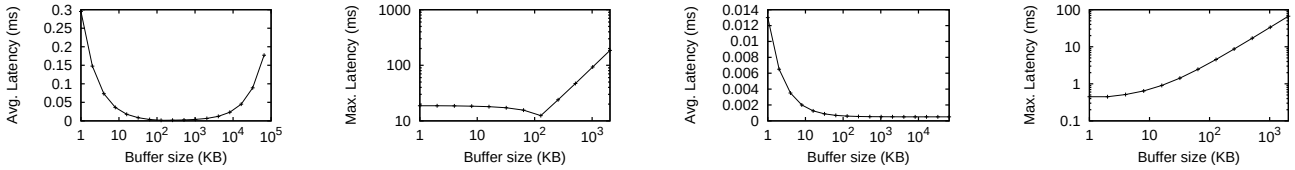
(*i*) What is the baseline performance of lookups and inserts in our design? How does the performance compare against existing disk-based indexes (e.g., the popular Berkeley-DB)? Are there specific workloads that our approach is best suited for?

(*ii*) To what extent do different optimizations in our design – namely, buffering of writes, use of bloom filters and use of bit-slicing – contribute towards our CLAM's overall performance?

(*iii*) To what extent does the use of flash-based secondary storage contribute to the performance?

(*iv*) How well does our design support a variety of hash table eviction policies?

We start by describing our implementation and how we configure it for our experiments.



(a) Avg. latency (chip) (b) Max. latency (chip) (c) Avg. latency (SSD) (d) Max. latency (SSD)

Figure 4: *Amortized and worst-case insertion cost on a flash chip and an Intel SSD. Only flash I/O costs are shown.*

7.1 Implementation and Configuration

We have implemented BufferHash in ∼3000 lines of C++ code. The hash table in a buffer is implemented using Cuckoo hashing [25] with two hash functions. Cuckoo hashing utilizes space efficiently and avoids the need for hash chaining in the case of collisions.

To simplify implementation, each partition is maintained in a separate file with all its incarnations. A new incarnation is written by overwriting the portion of file corresponding to the oldest incarnation in its super table. Thus, the performance numbers we report include small overheads imposed by the `ext3` file system we use. One can achieve better performance by writing directly to the disk as a raw device, bypassing the file system.

We run the BufferHash implementation atop two different SSDs: an Intel SSD (model: X18-M, which represents a new generation SSD), and a Transcend SSD (model: TS32GSSD25, which represents a relatively old generation but cheaper SSD).

7.1.1 Configuring the CLAM

As mentioned in §3, our key motivating applications like WAN optimization and deduplication employ hash tables of size 16-32GB. To match this, we configure our CLAMs with 32GB of slow storage and 4GB of DRAM. The size of a buffer in a super table is set to 128KB, as suggested by our analysis in §6.4. We limit the utilization of the hash table in a buffer to 50% as a higher utilization increases hash collision and the possibility of re-building the hash table for cuckoo hashing. Also, each hash entry takes 16 bytes of space. Thus, each buffer (and each incarnation) contains 4096 hash entries.

According to the analysis in §6.4, the optimal size of buffers for the above configuration is 266MB. We now experimentally validate this. Figure 5 shows the variation of false positive rates as the memory allocated to buffers is varied from 128KB to 3072MB in our prototype. The overall trend is similar to that shown by our analysis in §6.4, with the optimal spurious rate of 0.0001 occurring at a 256MB net size of buffers. The small difference from our analytically-derived optimal of 266MB arises because our analysis does not restrict the optimal number of hash functions to be an integer.

Note that the spurious rate is low even at 2GB (0.01). We select this configuration – 2GB for buffers and 32GB for slow storage – as the candidate configuration for the rest of our experiments. This gives us 16 incarnations per buffer and total of 16,384 buffers in memory.

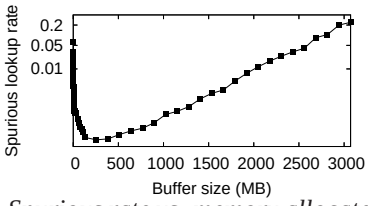


Figure 5: *Spurious rate vs. memory allocated to buffers; BufferHash configured with 4GB RAM and 32GB SSD.*

# of flash I/O	Probability		Latency (ms)	
	0% LSR	40% LSR	Flash chip	Intel SSD
0	0.9899	0.6032	0	0
1	0.0094	0.3894	0.24	0.31
2	0.0005	0.0073	0.48	0.62
3	0.00005	0.00003	0.72	0.93

Table 2: *A deeper look at lookup latencies.*

7.2 Lookups and Inserts

We start by considering the performance of basic hash table operations, namely lookups and inserts. We study other operations such as updates in §7.4.

We use synthetic workloads to understand the performance. Each synthetic workload consists of a sequence of lookups and insertions of keys. For simplicity, we focus on a single workload for the most part. In this workload, every key is first looked up, and then inserted. The keys are generated using random distribution with varying range; the range effects the lookup success rate (or, "LSR") of a key. These workloads are motivated by the WAN optimization application discussed in §8. We also consider other workloads with different ratio of insert and lookup operations in §7.2.3. Further, in order to stress-test our CLAM design, we assume that keys arrive in a continuous backlogged fashion in each workload.

7.2.1 Latencies

In Figure 6(a), we show the distribution of latencies for lookup operations on our CLAM with both an Intel and a Transcend SSD (the curves labeled **BH + SSD**) . This workload has an LSR of approximately 40%. Around 62% of the time, the lookups take little time (< 0.02ms) for both Intel and Transcend SSD, as they are served by either the in-memory bloom filters or in-memory buffers. 99.8% of the lookup times are less than 0.176ms for the

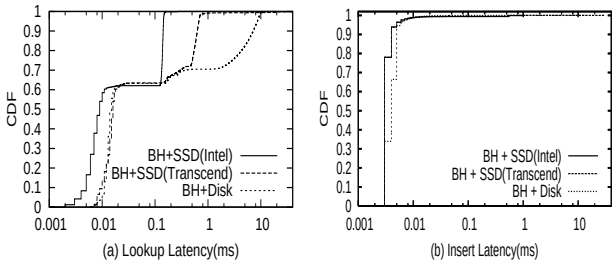


Figure 6: *CLAM latencies on different media.*

Intel SSD. For Transcend SSD, 90% of the lookup times are under 0.6ms, and the maximum is 1ms. The Intel SSD offers significantly better performance than Transcend SSD.

To understand the lookup latencies better, we examine the flash I/Os required by a lookup operation in our CLAM prototypes, under two different lookup success ratios in Table 2. Most lookups go to slow storage only when required, i.e., key is on slow storage (which happens in 0% of cases for LSR = 0 and in slightly under 40% of cases for LSR = 0.4); spurious flash I/O may be needed in the rare case of false positives (recall that BufferHash is configured for 0.01 false positive rate). Nevertheless, $>$ 99% lookups require at most one flash read only.

In Figure 6(b), we show the latencies for insertions on different CLAM prototypes. Since BufferHash buffers writes in memory before writing to flash, most insert operations are done in memory. Thus, the average insert cost is very small (0.006 ms and 0.007 ms for Intel and Transcend SSDs respectively). Since a buffer holds around 4096 items, only 1 out of 4096 insertions on average requires writing to flash. The worst case latency, when a buffer is flushed to the SSD and requires erasing a block, is 2.72 ms and 30 ms for Intel and Transcend SSDs, respectively.

On the whole, we note that our CLAM design achieves good lookup and insert performance by reducing unsuccessful lookups in slow storage and by batching multiple insertions into one big write.

7.2.2 Comparison with DB-Indexes

We now compare our CLAM prototypes against the hash table structure in Berkeley-DB (BDB) [6], a popular database index. We use the same workload as above. (We also considered the B-Tree index of BDB, but the performance was worse than the hash table. Results are omitted for brevity.) We consider the following system configurations: (1) **DB+SSD**: BDB running on an SSD, with BDB recommended configurations for SSDs, and (2) **DB+Disk**: BDB running on a magnetic disk.

Figures 7(a) and (b) show the lookup and the insert latencies for the two systems. The average lookup and insert latencies for **DB+Disk** are 6.8 ms and 7 ms respectively. More than 60% of the lookups and more than 40% of the inserts have latencies greater than 5 ms, corresponding to high seek cost on disks. Surprisingly, for the Intel SSD, the average lookup and insert latencies are also high – 4.6 ms and 4.8 ms respectively. Around 40% of lookups and 40% inserts have latencies greater than 5 ms! This is counterintuitive given that Intel SSD has significantly faster random I/O latency (0.15ms) than magnetic disks. This is explained by the fact that the low latency of an SSD is achieved only when the write load on the SSD is "low"; i.e., there are sufficient pauses between bursts of writes so that the SSD has enough time to clean dirty blocks to produce erased blocks for new writes [17]. Under a high write rate, the SSD quickly uses up its pool of erased blocks and then I/Os block until the SSD has reclaimed enough space from dirty blocks via garbage collection.

This result shows that existing disk based solutions that send all I/O requests to disks are not likely to perform well on SSDs, even if SSDs are significantly faster than disks (i.e., for workloads that give SSDs sufficient time for garbage collection). In other words, these solutions are not likely to exploit the performance benefit of SSDs under "high" write load. In contrast, since BufferHash writes to flash only when a buffer fills up, it poses a relatively "light" load on SSD, resulting in faster reads.

We do note that it is possible to supplement the BDB index with an in-memory Bloom filter to improve lookups. We anticipate that, on disks, a BDB with in-memory Bloom filter will have similar lookup latencies as a BufferHash. However, on SSDs, a BufferHash is likely to have a better lookup performance—because of the lack of buffering, insertions in BDB will incur a large number of small writes, which adversely affect SSDs' read performance due to fragmentation and background garbage collection activities.

7.2.3 Other Workloads

We evaluate how our CLAM design performs with workloads having different relative fractions of inserts and lookups. Our goal is to understand the workloads where the benefits of our design are the most significant. Table 3 shows the variation of the latency per operation with different lookup fractions in the workload for **BH+SSD** and **DB+SSD** on Transcend-SSD.

As Table 3 shows, the latency per operation for BDB decreases with increasing fraction of lookups. This is due to two reasons. First, (random) reads are significantly cheaper than random writes in SSDs. Since the increasing lookup fraction increases the overall fraction of flash reads, it reduces the overall latency per operation. Second, even the latency of individual lookups decreases with increasing fraction of lookups (not shown in the table). This is because, with a smaller fraction of flash

Fraction of lookups	Latency per operation (ms)	
	Bufferhash	Berkeley DB
0	0.007	18.4
0.3	0.01	13.5
0.5	0.09	10.3
0.7	0.11	5.3
1	0.12	0.3

Table 3: *Per-operation latencies with different lookup fractions in workloads. LSR=0.4 for all workloads and Transcend SSD is employed.*

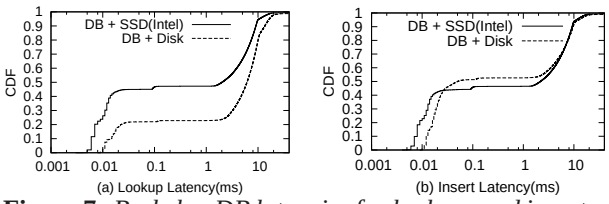


Figure 7: *Berkeley-DB latencies for lookups and inserts.*

write I/O, SSDs involve less garbage collection overhead that could interfere with all flash I/Os in general.

In contrast, for a CLAM, the latency per operation improves with decreasing fraction of lookups. This can be attributed to buffering due to which the average insert latency for a CLAM is reduced. As Table 3 shows, our CLAM design is 17× faster for write-intensive workloads than for read-intensive workloads.

7.3 Dissecting Performance Benefits

In what follows, we examine the contribution of different aspects of our CLAM design on the overall performance benefits it offers.

7.3.1 Contribution of BufferHash optimizations

The performance of our CLAM comes from three main optimizations within BufferHash: (1) buffering of inserts, (2) using Bloom filters, and (3) using windowed bit-slicing. To understand how much each of these optimizations contributes towards CLAM's performance, we evaluate our Intel SSD-based CLAM without one of these optimizations at a time.

The effect of buffering is obvious; without it, all insertions go to the flash, yielding an average insertion latency of ∼4.8ms at high insert rate i.e. continuous key insertions (compared to ∼0.006ms with buffering). Even at low insert rate, average insertion latency is ∼0.3ms and thus buffering gives significant benefits.

Without Bloom filters, each lookup operation needs to check many incarnations until the key is found or all incarnations have been checked. Since checking an incarnation involves a flash read, this makes lookups slower. The worst case happens with 0% redundancy, in which case each lookup needs to check all 16 incarnations. Our experiments show that even for 40% and 80% LSR, the average flash I/O latencies are 1.95ms and 1.5ms respectively without using Bloom filters. In contrast, using Bloom filters avoids expensive flash I/O, reducing flash I/O costs to 0.06ms and 0.13ms for 40% and 80% LSR respectively and giving a speedup of 10-30×.

Bit-slicing improves lookup latencies by ∼20% under low LSR, where the lookup workload is mostly memory bound. However, the benefit of using bit-slicing becomes negligible under a high LSR, since the lookup latency is then dominated by flash I/O latency.

7.3.2 Contribution of Flash-based Storage

The design of BufferHash is targeted specifically toward flash-based storage. In this section, we evaluate the contribution of the I/O properties of flash-based storage to the overall performance of our CLAM design. To aid in this, we compare two CLAM designs: (1) **BH+SSD**: BufferHash running on an SSD and (2) **BH+Disk**: BufferHash running on a magnetic disk (Hitachi Deskstar 7K80 drive). We use a workload with 40% look-up success rate over random keys with interleaved inserts and lookups.

Figures 6(a) and (b) also show the latencies for lookups and inserts in **BH+Disk**. Lookup latencies range from 0.1 to 12ms, an order of magnitude worse than the SSD prototypes (**BH+SSD**) due to the high seek latencies in disks. The average insert cost is very small and the worst case insert cost is 12ms, corresponding to a high seek latency for disk. Thus, the use of SSD contributes to the overall high performance of CLAM.

Comparing Figures 6 and 7, we see that **BH+Disk** performs better than **DB+SSD** and **DB+Disk** on both lookups and inserts. This shows that while using SSDs is important, it is not sufficient for high performance. It is crucial to employ BufferHash to best leverage the I/O properties of the SSDs.

7.4 Eviction Policies

Our experiments so far are based on the default FIFO eviction policy of BufferHash which we implement using the full discard primitive (§5.1.2). As stated earlier, the design of BufferHash is ideally suited for this policy. We now consider other eviction policies.

LRU. We implemented LRU using the full discard primitive as noted in §5.1.2. Omitting the details of our evaluation in the interest of brevity, we note that the performance of lookups was largely unaffected compared to FIFO; this is because the "re-insertion" operations that help emulate LRU happen asynchronously without blocking lookups (§5.1.2). In the case of inserts, the in-memory buffers get filled faster due to re-insertions, causing flushes to slower storage to become more frequent. The resulting increase in average insert latency, however, is very small: with a 40%-LSR workload having equal fractions of lookups and inserts, the average insert latency increases from 0.007ms to 0.008ms on Transcend SSD.

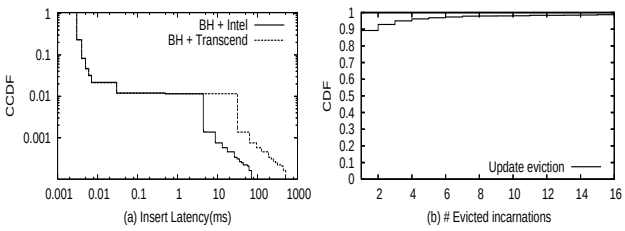


Figure 8: *(a) CCDF of insert latencies for the update-based policy. Both axes are in log-scale. (b) CDF of the number of incarnations tried upon a buffer flush. In both cases, the workload has 40% LSR and equal fractions of inserts and lookups.*

Partial discard. We now consider the two partial discard policies discussed in §5.1.2: the *update-based policy*, where only the stale entries are discarded, and the *priority-based policy*, where entries with priority lower than a threshold are discarded. We use a workload of 40% update rate using keys generated by random distribution. Figure 8(a) shows the CCDF of insert latencies on Transcend and Intel SSDs for the update-based policy. We note that an overwhelming fraction of the latencies remain unchanged, but the rest of the latencies (1%) worsen significantly. On the whole, this causes the average insertion cost to increase significantly to 0.56ms on Transcend SSD and 0.08ms on an Intel SSD. Nevertheless, this is still an order or magnitude smaller than the average latency when employing BerkeleyDB on SSDs. For priority-based policy, we used priority values equally distributed over keys. With different thresolds, we obtained similar qualitative performance (results omitted for brevity).

Three factors contribute to the higher latency observed in the tail of the distribution above: (1) When a buffer is flushed to slow storage, there is the additional cost of reading entries from the oldest incarnation and finding entries to be retained. (2) In the worst case, all entries in the evicted incarnation may have to be retained (for example, in the update-based policy this happens when none of the entries in the evicted incarnation have been updated or deleted). In that case, the in-memory buffer becomes full and is again flushed causing an eviction of the next oldest incarnation. These "cascaded evictions" continue until some entries in an evicted incarnation can be discarded, or all incarnations have been tried. In the latter case, all entries of the oldest incarnation are discarded. Cascaded evictions contribute to the high insertion cost seen in the tail of the distribution in Figure 8(a). (3) Since some of the entries are being retained after eviction, the buffer starts filling up more frequently and number of flushes to slow storage increases as a result.

We find that when buffer becomes full on a key insertion and needs to be flushed, the overall additional cost of insertion operations on Transcend SSD is 17.4ms (on average) for the update-based policy. Of this, the additional cost arising from reading and checking the entries of each incarnation is 1.62ms. Note that this is the only additional cost incurred when an incarnation eviction does not result in cascaded evictions. For priority-based policy, this cost is lower – 1.48 ms. The update-based policy is more expensive as it needs to search Bloom filters to see if an entry has already been updated.

We further find that only on rare occasions do cascaded evictions result in multiple incarnations getting accessed. In almost 90% of the cases where cascades happen, no more than 3 incarnations are tried as shown in Figure 8(b). On average, just 1.5 incarnations are tried (i.e., 0.5 incarnations are additionally flushed to slow storage, on average).

Thus, our approach supports FIFO and LRU eviction well, but it imposes a substantially higher cost for a small fraction of requests in other general eviction policies. The high cost can be controlled by loosening the semantics of the partial discard policies in order to limit cascaded evictions. For instance, applications using the priority-based policy could retain the top-k high priority entries rather than using a fixed threshold on priority. It is up to the application designer to select the right tradeoff between the semantics of the eviction policies and the additional overhead incurred.

7.5 Evaluation Summary

The above evaluation highlights the following aspects of our CLAM design:

(1) BufferHash on Intel SSD offers lookup latency of 0.06 ms and insert latency of 0.006 ms, and gives an order of magnitude improvement over Berkeley DB on Intel SSD.

(2) Buffering of writes significantly improves insert latency. Bloom filters significantly reduce unwanted lookups on slow storage, achieving $10\times$-$30\times$ improvement over BufferHash without bloom filters. Bit-slicing contributes 20% improvement when the lookup workload is mostly memory bound.

(3) For lookups, BufferHash on SSDs is an order of magnitude better than BufferHash on disk. However, SSDs alone are not sufficient to give high performance.

8 WAN Optimizer Using CLAM

In this section, we study the benefits of using our CLAM prototypes in an important application scenario, namely, WAN optimization.

A typical WAN optimizer has three components:

(1) Connection management (CM) front-end: When bytes from a connection arrive at the connection management front-end, they are accumulated into buffers for a short amount of time (we use 25ms). The buffered object

data is divided into chunks by computing content-based chunk boundaries using Rabin-Karp fingerprints [38, 34]. A SHA-1 hash is computed for each chunk thus identified.

(2) Compression engine (CE): The CE maintains a large content cache on a magnetic disk. SHA-1 fingerprints of cached content are stored in a large hash table. Fingerprints handed over by the CM are looked up in the hash table to identify similarity against prior content chunks. After redundancy has been identified, the incoming object is compressed and handed over to the network subsystem (described next). The object's chunks are inserted into the content cache in a serial fashion, and SHA-1 hashes for its chunks are inserted into the hash table with pointers to the on-disk addresses of corresponding chunks.

The CE's hash table can be stored either in a CLAM or using BDB on flash. The CLAM is configured with 4GB RAM and 32GB of Transcend SSD. The CLAM implements the full BufferHash functionality, including lazy updates with FIFO eviction as well as windowed bit slicing. For BDB-based WAN optimizer, we implement FIFO eviction from the hash table by maintaining an in-memory delete list of invalidated old hash table entries. The BDB hash table is also 32GB in size.

(3) Network sub-system (NS): The NS simply transmits the bytes handed over by CE over the outgoing network link. In commercial WAN optimizers, the NS uses an optimized custom TCP implementation that can send data at the highest possible rate (without needing repeated slow start, congestion avoidance etc.)

In order to focus on the efficacy of CE, we employ two simplifications in our evaluation: (1) We emulate a high-speed CM by pre-computing chunks and SHA-1 fingerprints for objects. (2) To emulate TCP optimization in NS, we simply use UDP to transmit data at close to link speed and turn off flow and control congestion control.

In our experiments, we vary the WAN link speed from 10Mbps to 0.5Gbps.

Evaluation: We use real packet traces in our evaluation. These traces were collected at University of Wisconsin-Madison's access link to the Internet and at the access link of a high volume Web server in the university. From these packet traces we construct object-level traces by grouping packets with the same connection 4-tuple into a single object and using an inactivity timeout of 10s. We also conducted thorough evaluation using a variety of synthetic traces where we varied the redundancy fraction. We omit the results for brevity and note that they are qualitatively similar.

Scenarios. We study two scenarios both based on replaying traces against our experimental setup:

(1) *Throughput test:* All objects arrive at once. We then measure the total time taken to transmit the objects with and without using our WAN optimizer. The ratio of the latter to the former measures the extent to which the WAN optimizer helps improve effective capacity of the attached WAN link, and we refer to it as the *effective bandwidth improvement*.

(2) *Acceleration under high load:* Here, objects arrive at a rate matching the link speed; thus, the link is 100% utilized when there is no compression. For each object, we measure the time difference from object arrival to the last byte of the object being sent, with and without WAN optimization. In either case, we also measure the throughput the object achieves (= effective size/time difference). When WAN optimization is used, the time difference includes the time to fingerprint the object, look for matches and compress the object. In addition, it may include delays due to earlier objects (e.g., updating the index with fingerprints for the earlier object). Finally, we measure the *per object throughput improvement* as the ratio of an object's throughput with and without the WAN optimizer.

8.1 Benefits of Using CLAMs

Scenario 1: Figure 9 shows the effective bandwidth improvement using CLAM-based and BDB based WAN-optimizers at different link speeds. Both WAN optimizers use Transcend-SSD. Figure 9(a) shows the results for a high (50%) redundancy trace (i.e., optimal improvement factor 2). The BDB-based WAN optimizer gives close-to-optimal improvement ($2\times$) at low link speeds of up to 10Mbps. However, at higher link speeds it becomes a bottleneck and drastically reduces the effective bandwidth instead of improving it. In comparison, CLAM-based WAN-optimizer gives close-to-ideal improvement at $10\times$ higher (100Mbps) link speeds and gives reasonable improvements even at 200 Mbps. It becomes a bottleneck at 400Mbps making its usage obsolete at such speeds. Using Intel-SSD, the CLAM-based WAN-optimizer can run up to 500 Mbps while offering close to ideal improvement, but using Intel-SSD with BDB does not improve the situation significantly. A similar trend was observed for the low (15%) redundancy trace (i.e., optimal improvement factor 1.18) whose results are shown in Figure 9(b). In this case, CLAM-based WAN-optimizer is able to operate at even higher link speeds while giving close to ideal improvement. This is because, when redundancy is low, lookups in the case of CLAMs seldom go to flash, which results in higher throughput.

Scenario 2: We fix the link speed to be 10Mbps for this analysis, because BDB is ineffective at higher speeds. We use a trace with 50% redundancy. We now take a closer look at the improvements by CLAMs and BDB. Figures 10 (a) and (b) show the relative throughput improvement on an object-by-object basis (Only improvements up to factor of 2 is shown). We see that

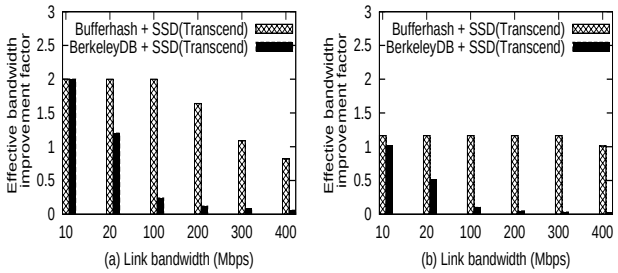


Figure 9: *Effective capacity improvement vs different link rates for (a) 50% and (b) 15% redundancy traces.*

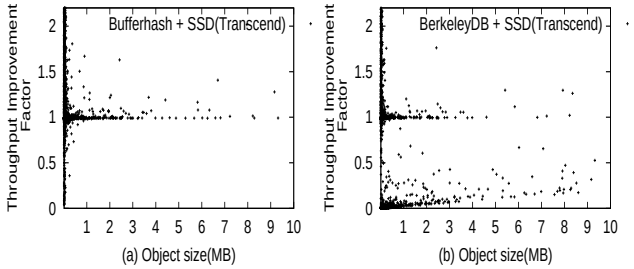


Figure 10: *Heavy load scenario: Throughput improvement per object for (a) BufferHash-based CLAM using Transcend SSD and (b) Berkeley-DB on Transcend SSD.*

Berkeley-DB has a negative effect on the throughputs of a large number of objects (compared to ideal), especially objects 500KB or smaller; their throughput is worsened by a factor of two or more due to the high costs of lookups and inserts (the latter for fingerprints of prior objects). Our CLAM also imposes overhead on some of these objects, but this happens on far fewer occasions and the overhead is significantly lower. Also, the average per-object improvement is 3.1 for our CLAM, which is 65% better than BDB (average improvement of 1.9).

9 Conclusions

We have designed and implemented CLAMs (Cheap and Large CAMs) for high-performance content-based networked systems that require large hash tables (up to 100GB or more) with support for fast insertion, lookups and updates. Our design uses a combination of DRAM and flash storage along with a novel data structure, called BufferHash, to facilitate fast hash table operations. Our CLAM supports a larger index than DRAM-only solutions, and faster hash operations than disk- or flash-only solutions. It can offer a few orders of magnitude more hash operations/s/$ than these alternatives. We have incorporated our CLAM prototype in a WAN optimizer and showed that it can enhance the benefits significantly.

Our design is not final, but it is a key step toward supporting high speed operation of modern data-intensive networked systems. It may be possible to design better CLAMs by leveraging space-saving ideas from recent systems such as FAWN [13] (to help control the amount of DRAM needed by BufferHash), using coding techniques such as floating codes (for better eviction support), or by using newer memory technologies such as Phase Change Memory (which can support much better read/write latencies than flash).

Acknowledgements. The authors would like the following people for their comments that helped improve our work directly or indirectly: Tom Anderson, Remzi Arpaci-Dusseau, Hari Balakrishnan, Flavio Bonomi, Patrick Crowley (our shepherd), Vivek Pai, Jennifer Rexford, Vyas Sekar, Srinivasan Seshan, Scott Shenker, Michael Swift and David Wetherall. This work is supported in part by an NSF FIND grant (CNS-0626889), an NSF CAREER Award (CNS-0746531), an NSF NetSE grant (CNS-0905134), and by grants from the UW-Madison Graduate School and Cisco.

References

[1] BlueCoat: WAN Optimization. `http://www.bluecoat.com/`.

[2] Cisco Wide Area Application Acceleration Services. `http://www.cisco.com/en/US/products/ps5680/Products_Sub_Category_Home.html`.

[3] Computerworld - WAN optimization continues growth. `www.computerworld.com.au/index.php/id;1174462047;fp;16;fpid;0/`.

[4] Disk Backup and deduplication with DataDomain. `http://www.datadomain.com`.

[5] Dropbox. `http://www.getdropbox.com`.

[6] Oracle Berkeley-DB. `http://www.oracle.com/technology/products/berkeley-db/index.html`.

[7] Peribit Networks (Acquired by Juniper in 2005): WAN Optimization Solution. `http://www.juniper.net/`.

[8] Riverbed Networks: WAN Optimization. `http://www.riverbed.com/solutions/optimize/`.

[9] Riverbed SteelHead Product Family. `http://www.riverbed.com/docs/SpecSheet-Riverbed-FamilyProduct_xx50_SMC-VE.pdf`.

[10] WAN Optimization Design. Private communication with a major vendor.

[11] D. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[12] A. Anand, S. Kappes, A. Akella, and S. Nath. Building cheap and large cams using bufferhash. Technical Report 1651, University of Wisconsin, 2009.

[13] D. Andersen, J. Franklin, M. Kaminsky, A. Phan-

ishayee, L. Tan, and V. Vasudevan. Fawn: A fast array of wimpy nodes. 2009.

[14] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26, 2003.

[15] L. Arge. The buffer tree: A new technique for optimal i/o-algorithms. In *4th International Workshop on Algorithms and Data Structures (WADS)*, 1995.

[16] A. Badam, K. Park, V. S. Pai, and L. Peterson. Hashcache: Cache storage for the next billion. In *NSDI*, 2009.

[17] L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding flash IO patterns. In *CIDR*, 2009.

[18] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP lookups. In *IEEE INFOCOM*, 2001.

[19] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2005.

[20] J. Bromley and W. Hutsell. Improve application performance and lower costs. Texas Instruments, http://www.texmemsys.com/files/f000240.pdf.

[21] A. Caulfield, L. Grupp, and S. Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS*, 2009.

[22] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD*, 2003.

[23] P. J. Desnoyers and P. Shenoy. Hyperion: high volume stream archival for retrospective querying. In *USENIX*, 2007.

[24] Endace Inc. Endace DAG3.4GE network monitoring card. `http://www.endace.com/`, 2009.

[25] U. Erlingsson, M. Manasse, and F. McSherry. A cool and practical alternative to traditional hash tables. In *Proceedings of the 7th Workshop on Distributed Data and Structures (WDAS'06)*, 2006.

[26] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems*, 2(4):267–288, 1984.

[27] H. Finucane, Z. Liu, and M. Mitzenmacher. Designing floating codes for expected performance. In *Allerton*, 2008.

[28] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *ACM SOSP*, 1987.

[29] G. Iannaccone, C. Diot, D. McAuley, A. Moore, I. Pratt, and L. Rizzo. The CoMo white paper. Technical Report IRC-TR-04-17, Intel Research, 2004.

[30] A. Jiang, V. Bohossian, and J. Bruck. Floating codes for joint information storage in write asymmetric memories. In *ISIT*, 2007.

[31] A. Kirsch and M. Mitzenmacher. The power of one move: Hashing schemes for hardware. In *IEEE INFOCOM*, 2008.

[32] T. Koponen, M. Chawla, B. Chun, A. Ermolinskiy, K. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *ACM SIGCOMM*, 2006.

[33] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *ACM SenSys*, 2006.

[34] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev.*, 35(5), 2001.

[35] S. Nath and P. B. Gibbons. Online Maintenance of Very Large Random Samples on Flash Storage . In *VLDB*, 2008.

[36] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *ACM/IEEE IPSN*, 2007.

[37] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *NSDI*, 2007.

[38] M. Rabin. Fingerprinting by random polynomials. Technical report, Harvard University, 1981. Technical Report, TR-15-81.

[39] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM SOSP*, 1991.

[40] StorageSearch. RAM SSDs versus Flash SSDs—technology and price trends. http://www.storagesearch.com/ssd-ram-v-flash.html.

[41] StreamBase Inc. Streambase: Real-time low latency data processing with a stream processing engine. `http://www.streambase.com/`, 2009.

[42] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[43] Z. Wei, K. Yi, and Q. Zhang. Dynamic external hashing: The limit of buffering. In *ACM SPAA*, 2009.

[44] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: an efficient index structure for fash-based sensor devices. In *USENIX FAST*, 2005.

[45] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST*, 2008.