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.

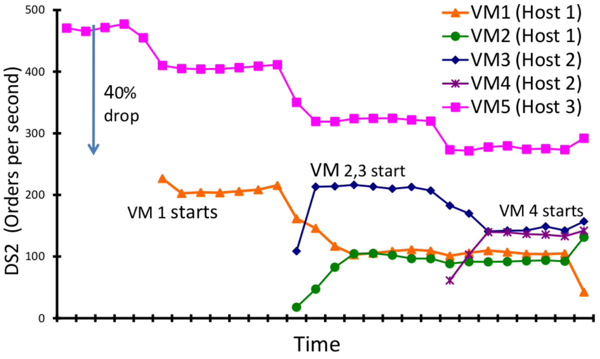


Figure 1: Orders/sec for VM5 decreases as the load on the shared storage device increases from VMs running on other hosts.

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 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].

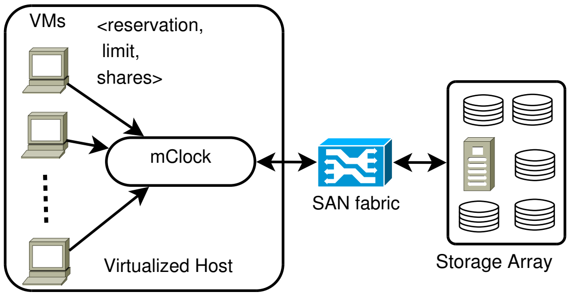


Figure 2: Virtualized host with VMs accessing a shared storage array over a SAN

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

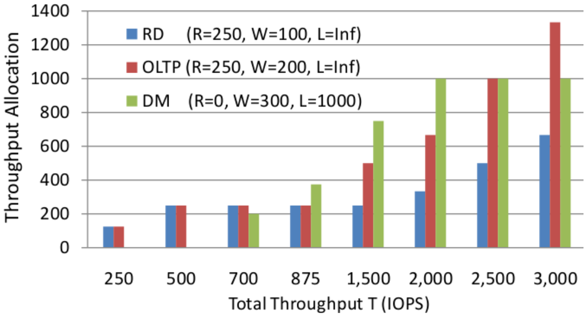


Figure 3: Allocation of IOPS to various VMs as the overall throughput changes

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:

$$R_i^r = \max\{R_i^{r-1} + 1/r_i,\ \mathit{Current\ time}\} \tag{3}$$

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 $\geq$ Max_QueueDepth then
        return;
    Let $E$ be the set of requests with $R$ tag $\leq$ 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 $\leq$ 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:

$$Lat = n(T_m + S/B_{peak}) \tag{4}$$

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:

$$Lat_2 = Lat_1 * (1 + \frac{S_2}{T_m \times B_{peak}})/(1 + \frac{S_1}{T_m \times B_{peak}}) \tag{5}$$

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:

$$IOPS = \text{Outstanding IOs}/Latency \tag{6}$$

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].

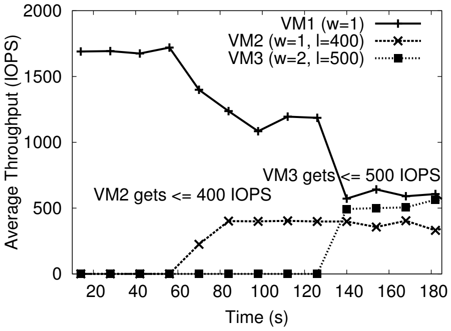
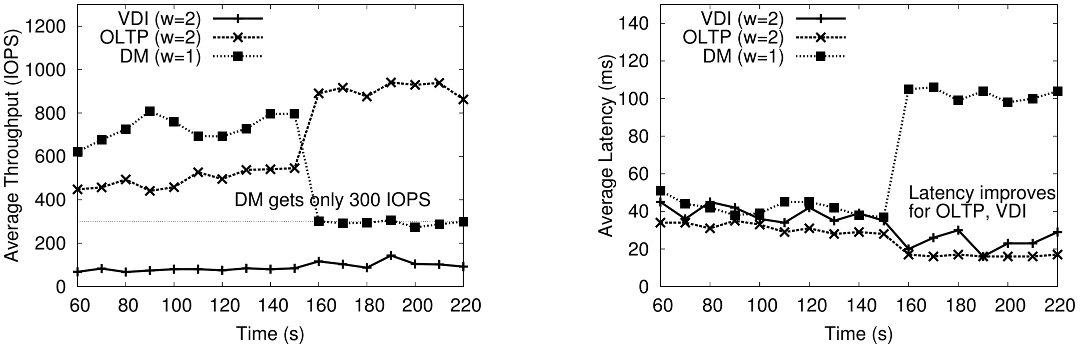


Figure 5: mClock limits the throughput of VM2 and VM3 to 400 and 500 IOPS as desired.

4.1.1 Limit Enforcement

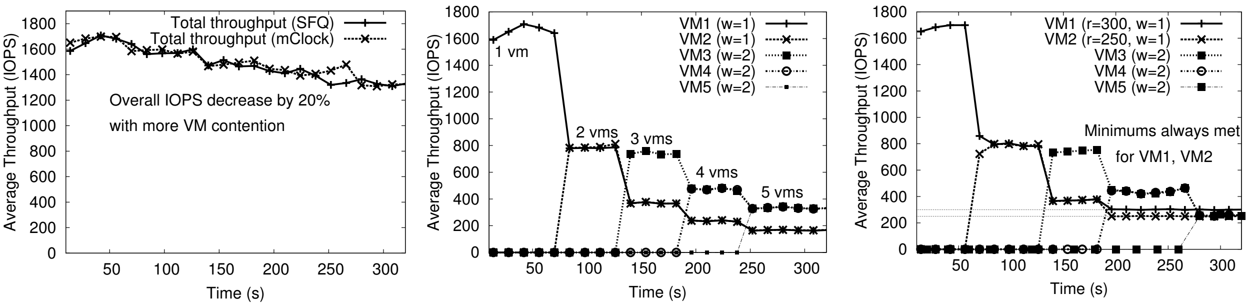
First we show the need for the limit control by demonstrating that pure proportional sharing cannot guarantee the specified number of IOPS and latency to a VM. We experimented with three workloads similar to those in the example of Section 2: RD, OLTP and DM.

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.



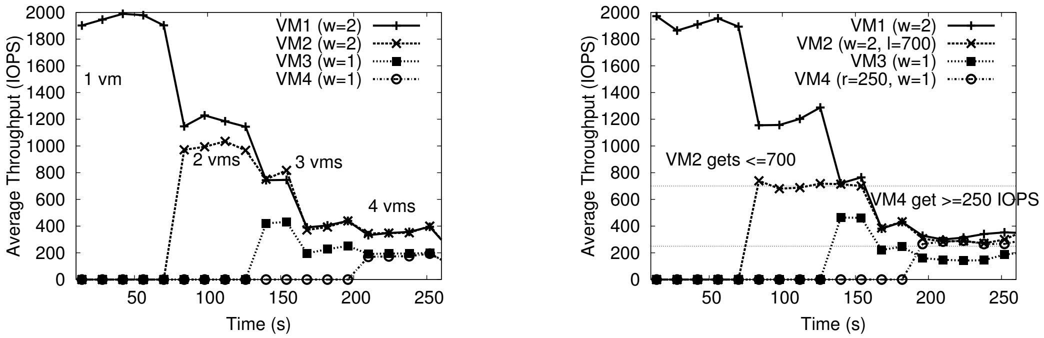
(a) Overall array throughput (b) SFQ (D) (c) mClock with r_1=300, r_2=250

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

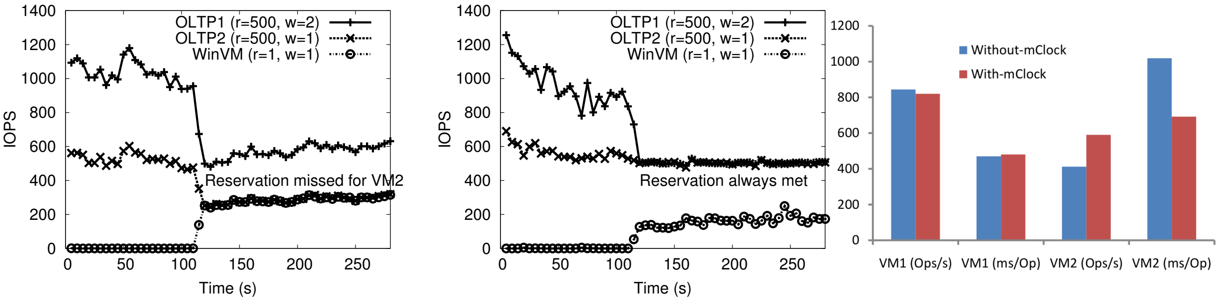
Table 3: VM workloads characteristics and parameters

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.

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.

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.

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 configured 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 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.

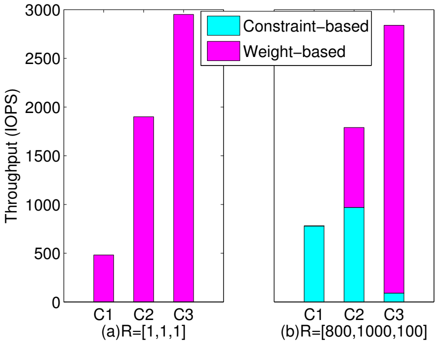


Figure 9: IOPS obtained by the three clients for two different cases. (a) All clients accessed the servers uniformly, with no reservations. (b) Clients had reservations of 800, 1000, and 100 IOPS, respectively.

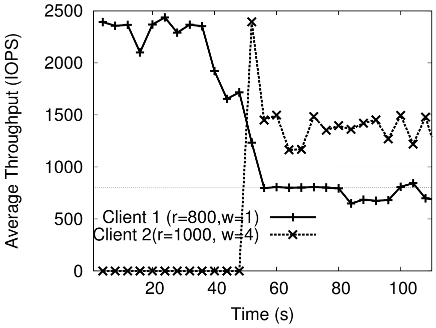


Figure 10: IOPS obtained by the two clients. When c_2 was started, c_1 still met its reservation target.

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.