mClock: Handling Throughput Variability for Hypervisor IO Scheduling

Ajay Gulati
VMware Inc
agulati@vmware.com

Arif Merchant
HP Labs
arif@hpl.hp.com

Peter Varman
Rice University
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 desired isolation using various resource management controls. Existing methods [3, 43] provide many knobs for allocating CPU and memory to VMs, but support for I/O resource allocation has been quite limited. IO resource management in a hypervisor introduces significant new challenges and needs more extensive controls compared to the 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 a minimum reservation and a maximum limit on the IO allocation for VMs. We present the design and implementation of mClock as a prototype inside VMware ESX hypervisor. Our results indicate that these rich set of QoS controls are quite effective in isolating VM performance and providing lower latencies to applications. We also show adaptation of mClock (called *dmClock*) to 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 an illusion of owning dedicated physical resources in terms of 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 provide guarantees for CPU and memory allocation using sophisticated controls such as reservations, limits and shares [43]. The current state of the art in terms of IO resource allocation is however much more rudimentary, and is limited to providing proportional shares [19] to different VMs.

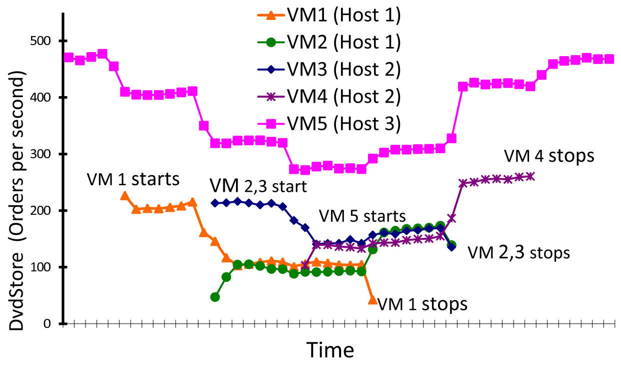


Figure 1: Highly fluctuating IOPS and orders/sec seen by VMs as the load on the shared storage changes.

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 is running one layer below the elevator based scheduling in the guest OS. So it needs to handle issues such as locality of accesses across VMs, high variability in IO sizes, different request priority based on the application running in a VM, and bursty workloads.

The amount of IO throughput available to any particular host can fluctuate widely based on the behavior of other hosts accessing the shared device. Consider a simple scenario shown in Figure 1 with 3 hosts and 5 VMs. Each VM is running a DvdStore [2] benchmark, which is an IO intensive OLTP workload. Initially VM 5 is running on host 3 and it achieves a transaction rate of roughly 500 orders/second. Later, as we start four other VMs (1-4) on two separate hosts sharing the same storage device, the throughput and transaction rate of the VMs decrease. The decrease is as much as 50% for VM 5, and the rate increases again as the VM's workload is stopped. Unlike CPU and memory resources, the IO throughput available to a host is not under its own control; instead changes in one host can affect the IO resources available to all other hosts. As shown above this can cause large variation in the IOPS available to a VM and impact application-level performance. 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 hosts and storage device.

PARDA [19] 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) [23] a traditional fair-scheduler, which divides the aggregate host throughput among the VMs in proportion to their shares. Unfortunately, when allocation is based on proportional shares alone, the absolute throughput for a VM can get diluted quickly as the throughput fluctuates. 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 cloud environments as well.

Resource controls such as *shares* (a.k.a. weights), *reservations*, and *limits* are used for predictable service allocation with strong isolation [8, 22, 42, 43]. 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 fixed and time-invariant, unlike the case with IO where the aggregate amount available to a host can vary with time as discussed above.

Shares are a relative allocation measure that specify the ratio in which the different VMs receive service. *Reservations* are used to provide a lower bound on absolute service allocation *e.g.* MHz for CPU or MBytes for memory. For similar reasons, it is desirable to provide a minimum reservation on the IO throughput of the VMs as well. When scheduling fixed-capacity resources the allocations to the VMs 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 this requirement is much more difficult when the

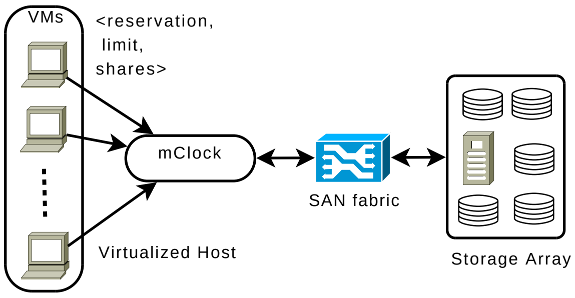


Figure 2: System Model

capacity fluctuates dynamically as in the case for shared IO. 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 at the end of every fixed time interval. This method, while conceptually simple, is not very effective in practice.

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 which cannot generally be sustained [1, 8].

Towards this goal, we present *mClock*, an IO scheduler that provides all three controls mentioned above at a per VM level. 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 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 [45], FAB [29]). *dmClock* ensures an overall allocation to each VM based on the specified shares, reservation, and limit parameters 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-

Algorithm class	**Proportional allocation**	**Latency control**	**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

specific optimizations. Distributed implementation for a clustered storage system is discussed in Section 3.2. Detailed performance evaluation using micro and macro-benchmarks 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 resources, such as Stonehenge [21] SFQ(D) [23], Argon [40] and Aqua [47]. Many of these algorithms are variants of weighted fair queuing mechanisms (Virtual Clock [49], WFQ [13], PGPS [28], WF^2Q [10], SCFQ [15], Leap Forward [37], SFQ [18] and Latency-rate scheduling [32]) proposed in the networking literature, but they 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 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 [27], BVT [14], pClock [20], Avatar [48] and service curve based techniques [12, 26, 30, 35]. Third is the class of algorithms that support reservation along with proportional allocation, such as Rialto [24], ESX memory management [43] and other reservation based CPU scheduling methods [17, 33, 34]. Table 1 provides a quick comparison of mClock with existing algorithms in 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 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 10000 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. with pending IOs $>$ 0). 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 example, the RD would receive its minimum 250 IOPS and the remaining 950 IOPS will be divided between OLTP and DM in the ratio 2 : 3, resulting in allocations of 380 and 570 IOPS respectively.

Table 2 shows the IOPS allocation to the three VMs in the example above, depending upon the current system throughput, T. If $T \geq 20000$ IOPS, then DM will be capped at 10000 IOPS because its share of $T/2$ is higher

VMs	RD	OLTP	DM
Weight	100	200	300
Reservation	250	250	0
Limit	∞	∞	10K
T (IOPS)	Allocations (IOPS)		
$0 \leq$ T ≤ 500	T/2	T/2	0
$500 \leq$ T ≤ 875	250	250	T - 500
$875 \leq$ T $\leq$ 1.5K	250	2(T-250)/5	3(T-250)/5
1.5K $\leq$ T $\leq$ 20K	T/6	T/3	T/2
T $\geq$ 20K	(T-10K)/3	2(T-10K)/3	10000

Table 2: Allocation based on different overall system's throughput values

than its upper limit, and the remainder is divided between RD and OLTP in the ratio 1 : 2. Similarly, if $T \leq 875$, RD and OLTP will each receive their reservations of 250 IOPS and the remainder will be allocated to DM. Finally, for $T \leq 500$ 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. Thee 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, 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 $T_P < 0$ the system throughput is insufficient to meet the reservations; in this case mClock simply gives each VM throughput proportional to its reservation. 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 array. 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.

2.2 Proportional Share Algorithms

A number of approaches such as Stonehenge [21], SFQ(D) [23], Argon [40] have been proposed for proportional sharing of storage between applications. Wang and Merchant [44] extended proportional sharing to distributed storage. Argon [40] and Aqua [47] propose service-time-based disk allocation to provide fairness as well as high efficiency. Brandt *et al.* [46] have proposed HDS that 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 [16,38] have been proposed but they mainly focus on CPU and memory resources and do not address variable capacity challenges like *mClock*.

2.3 Latency Supporting Algorithms

In the case of CPU scheduling, lottery scheduling [42, 36], BVT [14], and SMART [27] provide proportional allocation, latency-reducing mechanisms, and methods to handle priority inversion by exchanging tickets. Borrowed Virtual Time [14] and SMART [27] shift the virtual tag of latency-sensitive applications relative to the others to provide them with short-term advantage. pClock [20] and service-curve based methods [12,26,30, 35] 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, 33, 34]. 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

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 3: Symbols used and their descriptions

of MHz) 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 both reservation-based and proportional-share controls for both CPU and memory resources in a commercial product [8, 41, 43]. 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 [24] provide fixed reservations for known-capacity CPU service, while allowing additional service requests to be honored on an availability basis. Again Rialto requires re-computation of an allocation graph on each new arrival, which is then used for CPU scheduling.

3 mClock Algorithm

The intuitive idea behind the algorithm is to logically interleave a constraint-satisfying scheduler and a weight-based scheduler in a fine-grained manner. The constraint-satisfying scheduler ensures that VMs receive 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.

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.

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();
```

Tag Assignment: This routine assigns R, L and P tags to a request r from VM v_i arriving at time t. The R tag assigned to this request is at least $1/r_i$ beyond the last R tag value. However, if the current time is beyond 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. 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 roughly $T \times r_i$ requests with R tags in that interval. Similarly, the L tag is set to the maximum of the current time and $1/l_i + L_i^{r-1}$. 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.

The *proportional share tag* assigned to a request depends on the total number of requests of that VM that have completed service by the time it is dispatched. The proportional Share tag P_i^r is the larger of the arrival time of the request and $1/w_i + P_i^{r-1}$. Similar to the other tags, 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 gets active again. In virtual time based schedulers [10,15] this synchronization is done using global virtual time. Since the spacing of P tags is based on relative weights while the initial tag value is determined by the actual arrival time, there needs to be some mechanism to calibrate the tag values against real time. In the absence of such a mechanism starvation may occur, as tags of freshly active VMs will be unrelated to the existing tags. We adjust the origin of existing P tags to the current time, which is also the tag of a newly activated VM. In the implementation, an offset specifying the origin for each VM is adjusted and added to the tag value when used in scheduling. The relative ordering of existing tags is not altered by this transformation; however, newly-activated VMs start with a tag value equal to the smallest existing tag and, consequently, compete fairly with existing VMs.

Request Scheduling: As noted earlier, the scheduler alternates between constraint-satisfying 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 satisfying 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 [27]. 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-satisfying phase of the scheduler.

Request Type. mClock treats reads and writes identically. in practice writes show lower latency due to write buffering in the 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 tries to schedule all IOs within a VM in a FCFS manner 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 IOs with large sizes during tagging. The IO latency with n random outstanding IOs with an IO size of S each can be written as:

$$L = n(T_m + S/B_{peak}) \tag{3}$$

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 latency observed for an IO of size S_1 to a IO of a reference size S_2, keeping other factors constant would give:

$$L_2 = L_1 * (1 + \frac{S_2}{T_m \times B_{peak}})/(1 + \frac{S_1}{T_m \times B_{peak}}) \tag{4}$$

Using typical value of 5ms of mechanical delay and 60MB/s peak transfer rate, for a smaller reference size of 8KB, the numerator = $1 + 8/300 \approx 1$. So for tagging purposes, a single request of IO size S is treated 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 high degrees 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 (i.e. within 4 MB). Also the size of batch is bounded by a configurable parameter (set to 8). This optimization does impact the reservations to some extent mainly over short time intervals. 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. Allocating the benefit of locality to the concerned VM is part of future work.

Reservation Setting. Admission control is a well known and difficult problem for storage devices due to their stateful nature and dependence of throughput on workload. We propose a simple approach of using 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 get 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:

$$OIO = IOPS \times Latency \tag{5}$$

Thus for an application designed to keep 8 outstanding IOs, and requiring 25 ms average latency, the reservation should be set to 8 $\times$ 1000/25 = 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 [29], IBM's Intelligent Bricks [45]) 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-satisfying 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 also doesn't require any synchronization among 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 scheme avoids complex synchronization between the servers [31].

3.2.2 dmClock Comparison

To illustrate the behavior of *dmClock* and compare it with other existing approaches, we implemented *dmClock* and several other algorithms using a discrete event simulation environment.

Each storage server with capacity C services requests using an exponentially distributed service times with mean $1/C$. The clients can send requests to some or all of the servers. This simplified setup provides a convenient platform to compare various approaches and observe their behavior in a controlled environment. Note

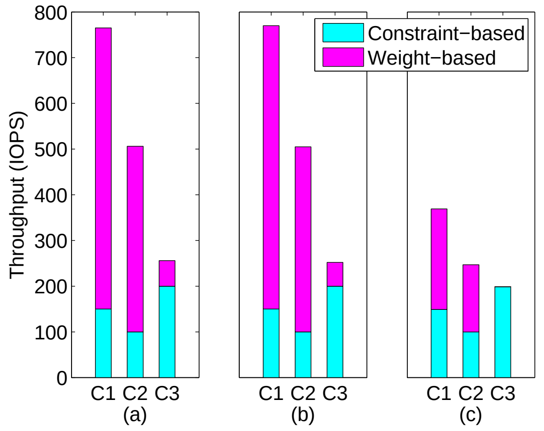


Figure 3: Distribution of requests done in constraint-based and weight-based phases by *mClock* for three cases. (a) All clients are uniformly accessing the servers. (b) Client 2 is hot-spotting on half of the servers. (c) Capacity is reduced to half.

that these simulation results here are only intended to gain insight and to compare *dmClock* with other algorithms. A more detailed performance evaluation of dmClock using an implementation in a real testbed is presented in Section 4.

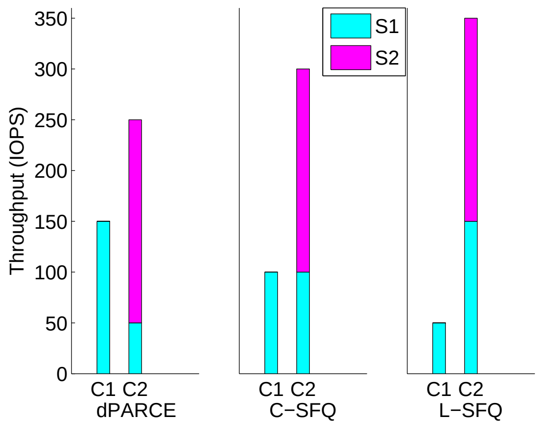


Figure 4: Comparison of *dmClock* with centralized SFQ and local SFQ running each server, when c_1 is hot-spotting at S_1

dmClock Allocation: We experimented with three clients (denoted c_1, c_2, c_3) and eight servers (denoted $S_1, S_2, ..., S_8$). Each server had an average capacity of 200 IOPS. The weights were set to 300, 200 and 100 for clients c_1, c_2 and c_3, respectively. The corresponding reservations were set to 100, 150 and 200 IOPS and the limits were set to infinity. These values were chosen to imitate a scenario where a client with a small share has a large reservation to obtain low latency. We tested three different cases, with uniform IO distribution across the servers, hot-spotting on one server, and variable capacity.

As mentioned earlier, *dmClock* does scheduling in two phases: constraint-based and weight-based. We looked at the distribution of requests done in each of these phases to get insight into the behavior. Figure 3(a) shows the distribution when all servers were accessed uniformly. Notice that the throughput during the constraint-based phase met the minimum requirement, and the rest of the requests were done during the weight-based phase. In this case clients received overall throughputs close to 750, 500 and 250, which is in proportion to their weights.

We ran another experiment with a similar capacity and requirements, but with c_1 hot-spotting by accessing only half of the servers. Figure 3(b) again shows the overall IOPS and the distribution of IOPS in two phases of the algorithm. Note that all three clients still meet their reservation and overall capacity is again allocated in ratio of their weights. At individual servers we observe that c_1 is getting more service at the servers at which it is hot-spotting. Also the service received by c_2 and c_3 increased in a proportional manner on the remaining servers.

Next, we changed the overall capacity by reducing the number of servers to four. In this case a proportional scheduler would have allocated about $800/6 = 133$ IOPS to client c_3, lower than its minimum requirement. Figure 3(c) shows that, using *dmClock*, client c_3 still met its reservation of 200 IOPS, and the remaining performance was divided between c_1 and c_2 as 350 and 250 IOPS, which is again in the proportion of their weights (3:2). These experiments show that *dmClock* always meets reservations (when possible) even in the distributed setting and uses any spare capacity to maintain global fairness.

3.2.3 Comparison With Other Approaches

Next, we compared *dmClock* with a centralized SFQ scheduler and with independent SFQ schedulers running at each server. Our experiment used a simple combination of two servers S_1 and S_2, and two clients c_1 and c_2. As in the previous experiment, the client c_1 represented a latency-sensitive workload that does not have a large throughput requirement. Accordingly, the ratio of weights was set to $1 : 3$ and the reservations were set to 150 and 50 IOPS for c_1 and c_2 respectively. The server capacity was set to 200 IOPS per server. To highlight the differences, we made c_1 hot spot, accessing only server S_1. We compared our algorithm with two other approaches: (1) C-SFQ: a centralized SFQ scheduler running as a shim appliance between all the clients and the servers, and (2) L-SFQ: independent, local SFQ schedulers, one on each server, each with weights in the ratio

1 : 3.

Figure 4 shows the average throughput obtained using *dmClock*, C-SFQ, and L-SFQ.

dmClock allocated about 150 IOPS to c_1, all from S_1, and the remaining IOPS (around 50 from S_1 plus 200 from S_2) to c_2 . S_2 was completely allocated to c_2 because there was no contention there. Thus, c_1 *received its reservation on* S_1 *and no more*. This shows that *dmClock* is globally fair while meeting reservations.

C-SFQ allocated the overall 100 and 300 IOPS to clients which is in ratio 1:3, causing c_1 to miss its reservation. This result is as we expect, since C-SFQ tries to be globally fair based on the weights, and does not use reservations.

L-SFQ allocated both servers in a locally fair (weight-proportional) manner by giving IOPS in ratio 1 : 3 at both servers. c_1 received only about 50 IOPS at S_1 and c_2 received the remaining 350 (150 + 200) IOPS available in the system. Thus, the global throughput distribution was neither proportional to the weights, nor did it meet the application minimum requirements. Again, this result is as expected, since L-SFQ neither uses information about global access pattern of different clients, nor does it use reservations.

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, 39]. We examine the following key questions about mClock in this evaluation: (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 I/O stack of VMware ESX server hypervisor to construct our prototype. The host is 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 other on a 10 disk, RAID 5 disk group. The host is configured to keep 32 IOs pending per LUN at the array, which is a typical setting.

We used a diverse set of workloads, using different operating systems, workload generators, and configurations, to test 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 VM operating system were on a different storage LUN. Three parameter 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, Read %, Randomness % and 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 [25].

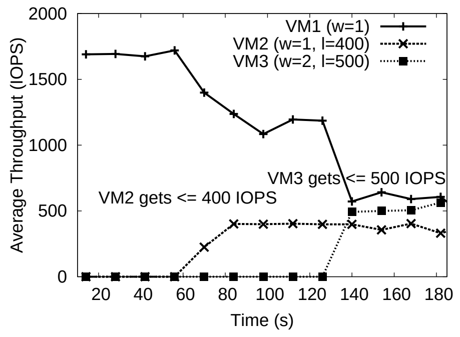
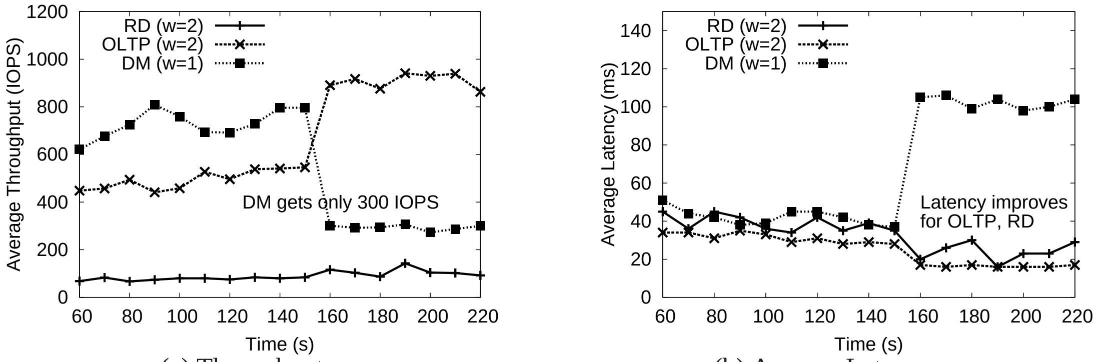


Figure 6: mClock limits the throughput of VM2 and VM3 to 400 and 500 IOPS as desired.

4.1.1 Limit Enforcement

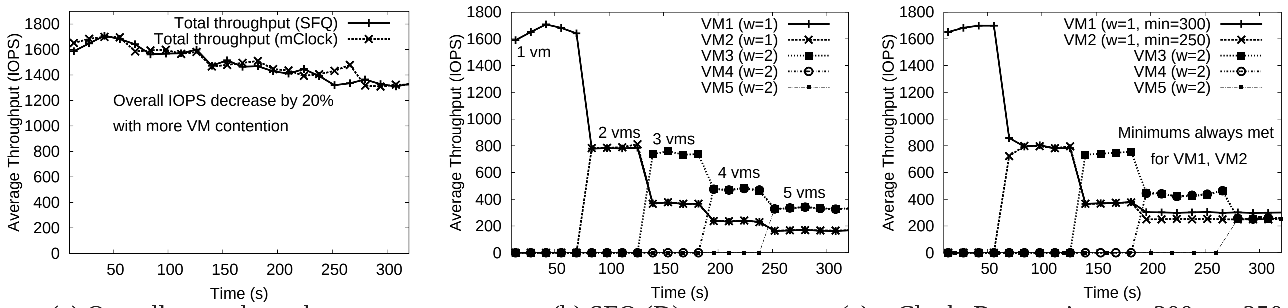
First we show the need for the limit control by demonstrating that pure proportional sharing cannot guarantee a specified number of IOPS and sufficiently low latency to a VM. We experimented with three workloads similar to those in the example of Section 2: RD, OLTP and DM.

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 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) [23]. The throughput and latency achieved is shown in Figures 5(a) and (b), between times zero and 140sec. Since RD is not fully backlogged, and OLTP has only 16 concurrent IOs, the



(a) Throughput (b) Average Latency

Figure 5: 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, Reservations r_1=300, r_2=250

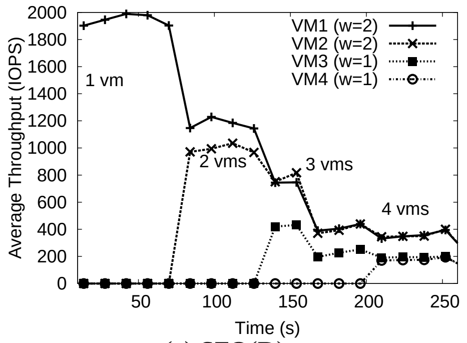
Figure 7: Five VMs with weights in ratio 1:1:2:2:2. VMs are started at 60 sec intervals. The overall throughput decreases as more VMs are added. mClock enforces reservations and SFQ only does proportional allocation.

work-conserving scheduler gives all the remaining queue slots (16 of them) to the DM workload. As a result, RD and OLTP get less than the specified proportion of IO throughput, while DM receives more. Since the device queue is always heavily occupied by IO requests from DM, the latency seen by RD and OLTP is 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 is that there are too many IOs from DM in the device queue.

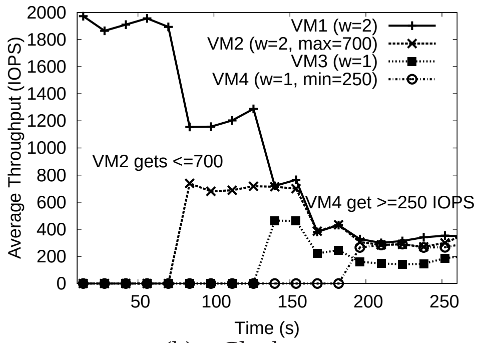
To provide better throughput and lower latency to RD and OLTP workloads, we changed the upper limit for DM to 300 IOs at $t = 140sec$. This caused the OLTP workload to see a 100% increase in throughput and the latency is reduced to half (36 ms to 16 ms). The RD workload also sees lower latency, while its throughput remains 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 diverse setting. 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 and 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 system has a capacity of approximately 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 VM1, VM2, and VM3 are running together.

Figure 6 shows the throughput obtained by each of the workloads. When we added the VM2 (at time 60sec), it only received 400 IOPS based on its limit, and not 800 IOPS which it would have gotten based on the weights alone. When we started VM3 (at time 120sec), it only received 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.



(a) SFQ(D)



(b) mClock

Figure 8: 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.

4.1.2 Reservations Enforcement

To test the ability of *mClock* in enforcing 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. Again to demonstrate the working of mClock in a dynamic environment, we begin with just VM1, and a new workload is started every 60 seconds.

Figures 7(a),(b) and (c) show the overall throughput observed by the host, and the throughput obtained for each workload using mClock. The throughputs using SFQ(D) with $D = 32$ are also shown as a baseline. 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. 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. This also led to better latency for VM1 and VM2 which would not have been possible just using proportional shares.

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 described in Table 4.

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 4: VM workloads characteristics and parameters

Figure 8(a) and (b) show the throughput allocated by SFQ(D) (weight-based allocation) and mClock for these workloads. mClock is able to restrict VM2 to 700 IOPS as desired when only two VMs are doing IOs. Later, when VM4 becomes active, mClock is able to meet the reservation of 250 IOPS for it, whereas SFQ only provides around 190 IOPS. While meeting these constraints, mClock is able to keep the allocation in proportion to the weights of the VMs; for example, VM1 gets twice as many IOPS as VM3 does.

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 the number of concurrent IOs from each application is 32. First, we 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 latencies received by each in this simultaneous run are shown in Table 5. 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 5. Note that first two VMs received higher IOPS of around 500 instead of 330 and 390, which allowed them to meet their reservations and also their latency tar-

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 5: Throughput and Latency observed by VMs running different workloads for $r_i = 1$ and 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 6: Throughput and Latency observed by VMs running different workloads for idle credit values 1 and 64

gets. The other VMs saw a corresponding decline in their throughput. This experiment demonstrates that mClock is able to provide a stronger 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 also that the reservations for all VMs have priority. 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 have equal shares, no reservation and no limit in terms of IOPS. We used idle-credit (σ) values of 1 and 64 for our experiment.

Table 6 shows the IOPS and average latency obtained by the bursty VM for the two settings of the idle credit. The number of IOPS is almost equal in either case because idle credits do not impact the overall bandwidth allocation over time, and VM1 has a bounded request rate. VM2 also sees almost the same IOPS for the two settings of idle credits. However, we notice that the latency seen by the bursty VM1 decreases as we increase the idle credits. VM2 also sees similar or slightly smaller latency, perhaps due to the increase in efficiency of doing a 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 with an upper bound of 256 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

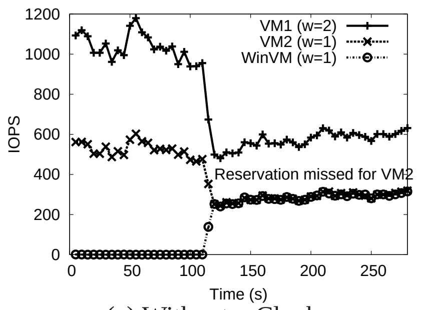
To test mClock with more realistic workloads, we experimented with two Linux RHEL VMs running OLTP workload using Filebench [25]. Each VMs is configured with 1 vCPU, 512 MB of RAM, 10GB database disk and 1 GB log virtual disk. To introduce throughput fluctuation another Windows 2003 VM running Iometer was used. The Iometer produced 32 concurrent, 16KB random reads. We assigned the weights in the ratio 2:1:1 to the two OLTP 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 9(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, VM2 started missing its reservation and received around 250 IOPS. When run with mClock, both the OLTP workloads were able to achieve their reservation of 500 IOPS. This shows that mClock can protect critical workloads from a sudden change in available throughput. The application-level metrics - number of operations/sec and transaction latency as reported by Filebench are summarized in Figure 9(c). Note that mClock was able to keep the latency of OLTP workload (480 ms) in VM2 from increasing, even with an increase in the overall IO contention.

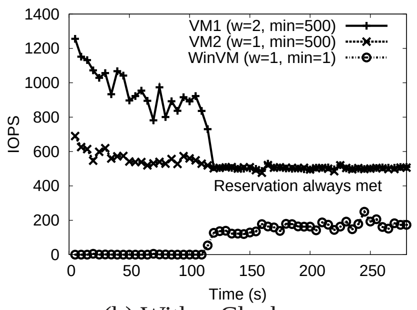
4.2 dmClock Evaluation

In this section, we present results of *dmClock* implementation in a distributed storage system. The system consists of multiple storage servers (nodes). Each node is implemented using a virtual machine running RHEL Linux with a 10GB OS disk and a 10GB experimental disk. Each experimental disk is placed on a different RAID-5 group with six disks, on a storage array. A single storage device is constructed using all storage nodes with data serviced from the local experimental disks. This represents a clustered-storage system where multiple storage nodes have dedicated storage devices that are used for servicing IOs. We used three storage servers for our experiment. Each experimental disk provided about 1500 IOPS for a random workload and the storage device is striped over 3 such experimental disks.

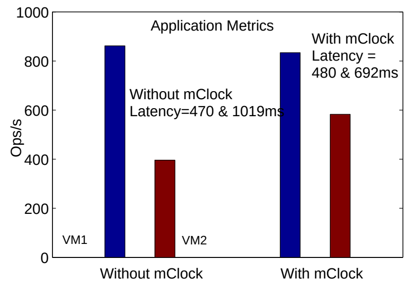
We implemented *dmClock* as a user-space module in each server node. The module receives IO requests containing IO size, offset, type (read/write) and the δ and ρ parameters, and data in the case of write requests.



(a) Without mClock



(b) With mClock



(c) Application metrics

Figure 9: (a) Without mClock, VM2 misses its minimum requirement when WinVM starts (b) With mClock, both OLTP workloads get their reserved IOPS despite WinVM workload (c) Application-level metrics: ops/s, avg Latency

The module can keep up to 16 outstanding IOs (using 16 server 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, piggy-backing 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. Here we have used our own workload generator because of the need to add appropriate δ and ρ values with each request.

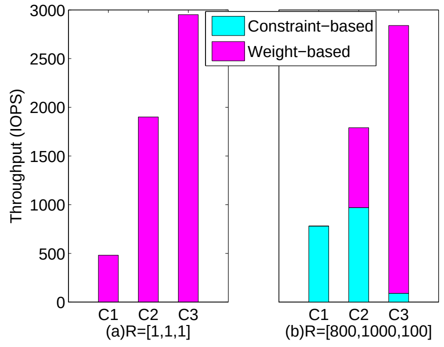


Figure 10: IOPS obtained by the three clients for two different cases. (a) All clients are uniformly accessing the servers, with no reservation. (b) Clients have reservation of 800, 1000 and 100 respectively.

In first experiment, we used three clients, $\{c_1, c_2, c_3\}$, each accessing all three server nodes. The weights were set in ratio 1:4:6, with no upper limit on 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 are used to highlight a use case where allocation based on reservations may be higher as compared to allocation based on weights or shares for some clients. The output for

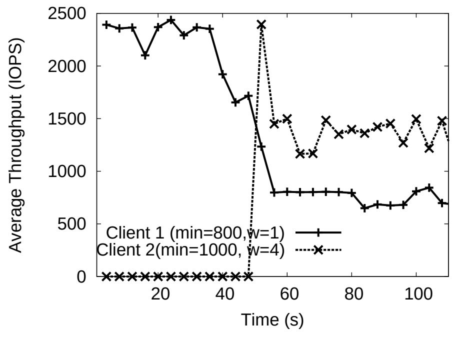


Figure 11: IOPS obtained by the two clients. Once c_2 is started, c_1 still meets its reservation target.

these two cases is shown in Figure 10 (a) and (b). Case (a) shows the overall IO throughput obtained by three clients without reservations. As expected, each client receives total service in proportion to its weight. In case (b), *dmClock* is 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 is divided among clients c_2 and c_3 in the ratio 2:3 as they respectively receive around 1750 and 2700 IOPS.

Next, we experimented with non-uniform accesses from clients. In this case we used two clients c_1 and c_2. 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 11 shows the IOPS obtained by the two clients with time. Note that initially c_1 gets the full capacity from server S_1 and when c_2 is started, c_1 is still able to meet its reservation of 800 IOPS. The remaining capacity is allocated to c_2, which received around 1400 IOPS. A distributed weight-proportional scheduler [44] 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 experiment 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 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 are expressed as minimum reservation, a maximum limit and proportional shares. The key aspect of mClock is its ability to enforce such control even in presence of fluctuating capacity, as shown by our implementation in the VMware ESX hypervisor. We also presented *dmClock*, a distributed version of our algorithm that can be used in clustered storage system architectures (such as FAB and IceCube). 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.

We believe that the controls provided by *mClock* would allow stronger isolation among VMs. Although we have shown the effectiveness for hypervisor IO scheduling, we think 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.

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, W. V. Courtright, II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa minor: versatile cluster-based storage. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.

[10] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *INFOCOM (1)*, 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*, 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*, New York, NY, USA, 1999. ACM.

[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, I. Ahmad, and C. Waldspurger. Parda: Proportional allocation of resources in distributed storage access. In *Usenix FAST*, Feb 2009.

[20] A. Gulati, A. Merchant, and P. Varman. *p*Clock: An arrival curve based approach for QoS in shared storage systems. In *ACM Sigmetrics*, 2007.

[21] L. Huang, G. Peng, and T. cker Chiueh. Multi-dimensional storage virtualization. In *SIGMETRICS '04*, pages 14–24, New York, NY, USA, 2004. ACM Press.

[22] H. A.-w. Ion Stoica 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.

[23] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *ACM SIGMETRICS '04*, 2004.

[24] M. B. Jones, D. Rosu, and M.-C. Rosu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *SOSP*. ACM, 1997.

[25] R. McDougall. Filebench: Application level file system benchmark. http://www.solarisinternals.com/si/tools/filebench/index.php.

[26] 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.

[27] J. Nieh and M. S. Lam. A smart scheduler for multimedia applications. *ACM Trans. Comput. Syst.*, 21(2):117–163, 2003.

[28] 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.

[29] Y. Saito et al. FAB: building distributed enterprise disk arrays from commodity components. *SIGPLAN Not.*, 39(11):48–58, 2004.

[30] 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.

[31] R. Stanojevic and R. Shorten. Fully decentralized emulation of best-effort and processor sharing queues. In *ACM SIGMETRICS*, 2008.

[32] 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.

[33] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the duality between resource reservation and proportional-share resource allocation. *SPIE*, February 1997.

[34] 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 *In Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288–299, 1996.

[35] 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.

[36] D. G. Sullivan and M. I. Seltzer. Isolation with flexibility: a resource management framework for central servers. In *USENIX Annual Technical Conference*, 2000.

[37] 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.

[38] 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.

[39] VMware, Inc. *Introduction to VMware Infrastructure*. 2007. http://www.vmware.com/support/pubs/.

[40] 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.

[41] C. Waldspurger. Personal Communications.

[42] C. A. Waldspurger. *Lottery and stride scheduling: flexible proportional-share resource management*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.

[43] 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.

[44] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *Usenix FAST*, Feb 2007.

[45] W. Wilcke et al. IBM intelligent bricks project — petabytes and beyond. *IBM Journal of Research and Development*, 50, 2006.

[46] J. C. Wu, S. Banachowski, and S. A. Brandt. Hierarchical disk sharing for multimedia systems. In *NOSSDAV*. ACM, 2005.

[47] 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 209–18, May 2006.

[48] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. In *MASCOTS*, pages 135–142, 2005.

[49] L. Zhang. VirtualClock: A new traffic control algorithm for packet-switched networks. *ACM Trans. Comput. Syst.*, 9(2):101–124.