PARDA: Proportional Allocation of Resources for Distributed Storage Access

Ajay Gulati Irfan Ahmad Carl A. Waldspurger
VMware Inc.
{*agulati,irfan,carl*}*@vmware.com*

Abstract

Rapid adoption of virtualization technologies has led to increased utilization of physical resources, which are multiplexed among numerous workloads with varying demands and importance. Virtualization has also accelerated the deployment of shared storage systems, which offer many advantages in such environments. Effective resource management for shared storage systems is challenging, even in research systems with complete end-to-end control over all system components. Commercially-available storage arrays typically offer only limited, proprietary support for controlling service rates, which is insufficient for isolating workloads sharing the same storage volume or LUN.

To address these issues, we introduce PARDA, a novel software system that enforces proportional-share fairness among distributed hosts accessing a storage array, without assuming any support from the array itself. PARDA uses latency measurements to detect overload, and adjusts issue queue lengths to provide fairness, similar to aspects of flow control in FAST TCP. We present the design and implementation of PARDA in the context of VMware ESX Server, a hypervisor-based virtualization system, and show how it can be used to provide differential quality of service for unmodified virtual machines while maintaining high efficiency. We evaluate the effectiveness of our implementation using quantitative experiments, demonstrating that this approach is practical.

1 Introduction

Storage arrays form the backbone of modern data centers by providing consolidated data access to multiple applications simultaneously. Deployments of consolidated storage using Storage Area Network (SAN) or Network-Attached Storage (NAS) hardware are increasing, motivated by easy access to data from anywhere at any time, ease of backup, flexibility in provisioning, and centralized administration. This trend is further fueled by the proliferation of virtualization technologies, which rely on shared storage to support features such as live migration of workloads across hosts.

A typical virtualized data center consists of multiple physical hosts, each running several virtual machines (VMs). Many VMs may compete for access to one or more logical units (LUNs) on a single storage array. The resulting contention at the array for resources such as controllers, caches, and disk arms leads to unpredictable IO completion times. Resource management mechanisms and policies are required to enable performance isolation, control service rates, and enforce service-level agreements.

In this paper, we target the problem of providing coarse-grained fairness to VMs, without assuming any support from the storage array itself. We also strive to remain work-conserving, so that the array is utilized efficiently. We focus on proportionate allocation of IO resources as a flexible building block for constructing higher-level policies. This problem is challenging for several reasons, including the need to treat the array as an unmodifiable black box, unpredictable array performance, uncertain available bandwidth, and the desire for a scalable decentralized solution.

Many existing approaches [13, 14, 16, 21, 25, 27, 28] allocate bandwidth among multiple applications running on a single host. In such systems, one centralized scheduler has complete control over all requests to the storage system. Other centralized schemes [19, 30] attempt to control the queue length at the device to provide tight latency bounds. Although centralized schedulers are useful for host-level IO scheduling, in our virtualized environment we need an approach for coordinating IO scheduling across multiple independent hosts accessing a shared storage array.

More decentralized approaches, such as Triage [18], have been proposed, but still rely on centralized measurement and control. A central agent adjusts per-host bandwidth caps over successive time periods and communicates them to hosts. Throttling hosts using caps can lead to substantial inefficiency by under-utilizing array resources. In addition, host-level changes such as VMs becoming idle need to propagate to the central controller, which may cause a prohibitive increase in communication costs.

We instead map the problem of distributed storage access from multiple hosts to the problem of flow control in networks. In principle, fairly allocating storage bandwidth with high utilization is analogous to distributed hosts trying to estimate available network bandwidth and consuming it in a fair manner. The network is effectively a black box to the hosts, providing little or no information about its current

state and the number of participants. Starting with this loose analogy, we designed PARDA, a new software system that enforces coarse-grained proportional-share fairness among hosts accessing a storage array, while still maintaining high array utilization.

PARDA uses the IO latency observed by each host as an indicator of load at the array, and uses a control equation to adjust the number of IOs issued per host, *i.e.*, the host *window size*. We found that variability in IO latency, due to both request characteristics (*e.g.*, degree of sequentiality, reads vs. writes, and IO size) and array internals (*e.g.*, request scheduling, caching and block placement) could be magnified by the independent control loops running at each host, resulting in undesirable divergent behavior.

To handle such variability, we found that using the average latency observed across *all* hosts as an indicator of overall load produced stable results. Although this approach does require communication between hosts, we need only compute a simple average for a single metric, which can be accomplished using a lightweight, decentralized aggregation mechanism. PARDA also handles idle VMs and bursty workloads by adapting per-host weights based on long-term idling behavior, and by using a local scheduler at the host to handle short-term bursts. Integrating with a local proportional-share scheduler [10] enables fair end-to-end access to VMs in a distributed environment.

We implemented a complete PARDA prototype in the VMware ESX Server hypervisor [24]. For simplicity, we assume all hosts use the same PARDA protocol to ensure fairness, a reasonable assumption in most virtualized clusters. Since hosts run compatible hypervisors, PARDA can be incorporated into the virtualization layer, and remain transparent to the operating systems and applications running within VMs. We show that PARDA can maintain cluster-level latency close to a specified threshold, provide coarse-grained fairness to hosts in proportion to per-host weights, and provide end-to-end storage IO isolation to VMs or applications while handling diverse workloads.

The next section presents our system model and goals in more detail. Section 3 develops the analogy to network flow control, and introduces our core algorithm, along with extensions for handling bursty workloads. Storage-specific challenges that required extensions beyond network flow control are examined in Section 4. Section 5 evaluates our implementation using a variety of quantitative experiments. Related work is discussed in section 6, while conclusions and directions for future work are presented in Section 7.

2 System Model

PARDA was designed for distributed systems such as the one shown in Figure 1. Multiple hosts access one or more storage arrays connected over a SAN. Disks in storage arrays are partitioned into RAID groups, which are used to construct LUNs. Each LUN is visible as a storage device to hosts and exports a cluster filesystem for distributed access. A VM disk is represented by a file on one of the shared LUNs, accessible from multiple hosts. This facilitates migration of VMs between hosts, avoiding the need to transfer disk state.

Since each host runs multiple virtual machines, the IO traffic issued by a host is the aggregated traffic of all its VMs that are currently performing IO. Each host maintains a set of pending IOs at the array, represented by an *issue queue*. This queue represents the IOs scheduled by the host and currently pending at the array; additional requests may be pending at the host, waiting to be issued to the storage array. Issue queues are typically per-LUN and have a fixed maximum *issue queue length*[1] (*e.g.*, 64 IOs per LUN).

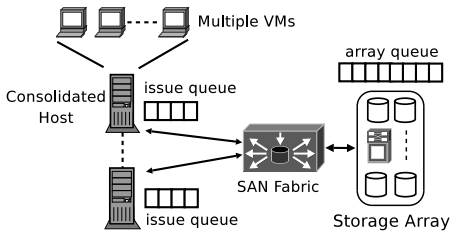


Figure 1: Storage array accessed by distributed hosts/VMs.

IO requests from multiple hosts compete for shared resources at the storage array, such as controllers, cache, interconnects, and disks. As a result, workloads running on one host can adversely impact the performance of workloads on other hosts. To support performance isolation, resource management mechanisms are required to specify and control service rates under contention.

Resource allocations are specified by numeric *shares*, which are assigned to VMs that consume IO resources.[2] A VM is entitled to consume storage array resources proportional to its share allocation, which specifies the relative importance of its IO requests compared to other VMs. The IO shares associated with a host is simply the total number of per-VM shares summed across all of its VMs. Proportional-share fairness is defined as providing storage array service to hosts in proportion to their shares.

In order to motivate the problem of IO scheduling across multiple hosts, consider a simple example with four hosts running a total of six VMs, all accessing a common shared LUN over a SAN. Hosts 1 and 2 each run two Linux VMs configured with OLTP workloads using Filebench [20].

[1]The terms *queue length*, *queue depth*, and *queue size* are used interchangeably in the literature. In this paper, we will also use the term *window size*, which is common in the networking literature.

[2]Shares are alternatively referred to as *weights* in the literature. Although we use the term *VM* to be concrete, the same proportional-share framework can accommodate other abstractions of resource consumers, such as applications, processes, users, or groups.

Host	VM Types	s_1, s_2	$VM1$	$VM2$	T_h
1	2×OLTP	20, 10	823 Ops/s	413 Ops/s	1240
2	2×OLTP	10, 10	635 Ops/s	635 Ops/s	1250
3	1×Micro	20	710 IOPS	n/a	710
4	1×Micro	10	730 IOPS	n/a	730

Table 1: Local scheduling does not achieve inter-host fairness. Four hosts running six VMs without PARDA. Hosts 1 and 2 each run two OLTP VMs, and hosts 3 and 4 each run one micro-benchmark VM issuing 16 KB random reads. Configured shares (s_i), Filebench operations per second (Ops/s), and IOPS (T_h for hosts) are respected within each host, but not across hosts.

Hosts 3 and 4 each run a Windows Server 2003 VM with Iometer [1], configured to generate 16 KB random reads. Table 1 shows that the VMs are configured with different share values, entitling them to consume different amounts of IO resources. Although a local start-time fair queuing (SFQ) scheduler [16] does provide proportionate fairness within each individual host, per-host local schedulers alone are insufficient to provide isolation and proportionate fairness *across* hosts. For example, note that the aggregate throughput (in IOPS) for hosts 1 and 2 is quite similar, despite their different aggregate share allocations. Similarly, the Iometer VMs on hosts 3 and 4 achieve almost equal performance, violating their specified 2 : 1 share ratio.

Many units of allocation have been proposed for sharing IO resources, such as Bytes/s, IOPS, and disk service time. Using Bytes/s or IOPS can unfairly penalize workloads with large or sequential IOs, since the cost of servicing an IO depends on its size and location. Service times are difficult to measure for large storage arrays that service hundreds of IOs concurrently.

In our approach, we conceptually partition the array queue among hosts in proportion to their shares. Thus two hosts with equal shares will have equal queue lengths, but may observe different throughput in terms of Bytes/s or IOPS. This is due to differences in per-IO cost and scheduling decisions made within the array, which may process requests in the order it deems most efficient to maximize aggregate throughput. Conceptually, this effect is similar to that encountered when time-multiplexing a CPU among various workloads. Although workloads may receive equal time slices, they will retire different numbers of instructions due to differences in cache locality and instruction-level parallelism. The same applies to memory and other resources, where equal hardware-level allocations do not necessarily imply equal application-level progress.

Although we focus on issue queue slots as our primary fairness metric, each queue slot could alternatively represent a fixed-size IO operation (*e.g.*, 16 KB), thereby providing throughput fairness expressed in Bytes/s. However, a key benefit of managing queue length instead of throughput is that it automatically compensates workloads with lower per-IO costs at the array by allowing them to issue more requests. By considering the actual cost of the work performed by the array, overall efficiency remains higher.

Since there is no central server or proxy performing IO scheduling, and no support for fairness in the array, a per-host *flow control* mechanism is needed to enforce specified resource allocations. Ideally, this mechanism should achieve the following goals: (1) provide coarse-grained proportional-share fairness among hosts, (2) maintain high utilization, (3) exhibit low overhead in terms of per-host computation and inter-host communication, and (4) control the overall latency observed by the hosts in the cluster.

To meet these goals, the flow control mechanism must determine the maximum number of IOs that a host can keep pending at the array. A naive method, such as using static per-host issue queue lengths proportional to each host's IO shares, may provide reasonable isolation, but would not be work-conserving, leading to poor utilization in underloaded scenarios. Using larger static issue queues could improve utilization, but would increase latency and degrade fairness in overloaded scenarios.

This tradeoff between fairness and utilization suggests the need for a more dynamic approach, where issue queue lengths are varied based on the current level of contention at the array. In general, queue lengths should be increased under low contention for work conservation, and decreased under high contention for fairness. In an equilibrium state, the queue lengths should converge to different values for each host based on their share allocations, so that hosts achieve proportional fairness in the presence of contention.

3 IO Resource Management

In this section we first present the analogy between flow control in networks and distributed storage access. We then explain our control algorithm for providing host-level fairness, and discuss VM-level fairness by combining cluster-level PARDA flow control with local IO scheduling at hosts.

3.1 Analogy to TCP

Our general approach maps the problem of distributed storage management to flow control in networks. TCP running at a host implements flow control based on two signals from the network: round trip time (RTT) and packet loss probability. RTT is essentially the same as IO request latency observed by the IO scheduler, so this signal can be used without modification.

However, there is no useful analog of network packet loss in storage systems. While networking applications expect dropped packets and handle them using retransmission, typical storage applications do not expect dropped IO requests, which are rare enough to be treated as hard failures.

Thus, we use IO latency as our only indicator of congestion at the array. To detect congestion, we must be able to distinguish underloaded and overloaded states. This is accomplished by introducing a *latency threshold* parameter, denoted by $\mathscr{L}$. Observed latencies greater than $\mathscr{L}$ may trigger a reduction in queue length. FAST TCP, a recently-proposed variant of TCP, uses packet latency instead of packet loss probability, because loss probability is difficult to estimate accurately in networks with high bandwidth-delay products [15]. This feature also helps in high-bandwidth SANs, where packet loss is unlikely and TCP-like AIMD (additive increase multiplicative decrease) mechanisms can cause inefficiencies. We use a similar adaptive approach based on average latency to detect congestion at the array.

Other networking proposals such as RED [9] are based on early detection of congestion using information from routers, before a packet is lost. In networks, this has the added advantage of avoiding retransmissions. However, most proposed networking techniques that require router support have not been adopted widely, due to overhead and complexity concerns; this is analogous to the limited QoS support in current storage arrays.

3.2 PARDA Control Algorithm

The PARDA algorithm detects overload at the array based on average IO latency measured over a fixed time period, and adjusts the host's issue queue length (*i.e.*, window size) in response. A separate instance of the PARDA control algorithm executes on each host.

There are two main components: latency estimation and window size computation. For latency estimation, each host maintains an exponentially-weighted moving average of IO latency at time t, denoted by $L(t)$, to smooth out short-term variations. The weight given to past values is determined by a smoothing parameter $\alpha \in [0,1]$. Given a new latency observation l,

$$L(t) = (1-\alpha) \times l \ + \ \alpha \times L(t-1) \tag{1}$$

The window size computation uses a control mechanism shown to exhibit stable behavior for FAST TCP:

$$w(t+1) = (1-\gamma)w(t) \ + \ \gamma\left(\frac{\mathscr{L}}{L(t)}w(t) + \beta\right) \tag{2}$$

Here $w(t)$ denotes the window size at time t, $\gamma \in [0,1]$ is a smoothing parameter, $\mathscr{L}$ is the system-wide latency threshold, and β is a per-host parameter that reflects its IO shares allocation.

Whenever the average latency $L > \mathscr{L}$, PARDA decreases the window size. When the overload subsides and $L < \mathscr{L}$, PARDA increases the window size. Window size adjustments are based on latency measurements, which indicate load at the array, as well as per-host β values, which specify relative host IO share allocations.

To avoid extreme behavior from the control algorithm, $w(t)$ is bounded by $[w_{min}, w_{max}]$. The lower bound w_{min} prevents starvation for hosts with very few IO shares. The upper bound w_{max} avoids very long queues at the array, limiting the latency seen by hosts that start issuing requests after a period of inactivity. A reasonable upper bound can be based on typical queue length values in uncontrolled systems, as well as the array configuration and number of hosts.

The latency threshold $\mathscr{L}$ corresponds to the response time that is considered acceptable in the system, and the control algorithm tries to maintain the overall cluster-wide latency close to this value. Testing confirmed our expectation that increasing the array queue length beyond a certain value doesn't lead to increased throughput. Thus, $\mathscr{L}$ can be set to a value which is high enough to ensure that a sufficiently large number of requests can always be pending at the array. We are also exploring automatic techniques for setting this parameter based on long-term observations of latency and throughput. Administrators may alternatively specify $\mathscr{L}$ explicitly, based on cluster-wide requirements, such as supporting latency-sensitive applications, perhaps at the cost of under-utilizing the array in some cases.

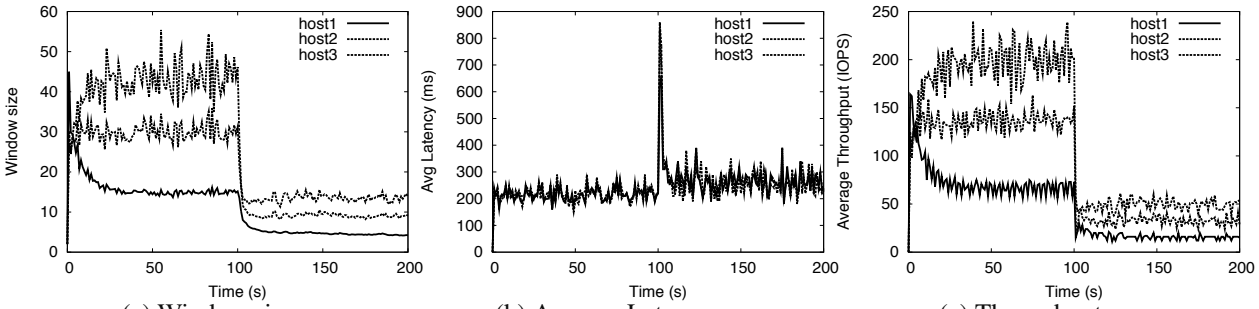
Finally, β is set based on the IO shares associated with the host, proportional to the sum of its per-VM shares. It has been shown theoretically in the context of FAST TCP that the equilibrium window size value for different hosts will be proportional to their β parameters [15].

We highlight two properties of the control equation, again relying on formal models and proofs from FAST TCP. First, at equilibrium, the throughput of host i is proportional to β_i/q_i, where β_i is the per-host allocation parameter, and q_i is the queuing delay observed by the host. Second, for a single array with capacity C and latency threshold $\mathscr{L}$, the window size at equilibrium will be:

$$w_i = \beta_i \ + \ \beta_i \ \frac{C\mathscr{L}}{\sum_{\forall j} \beta_j} \tag{3}$$

To illustrate the behavior of the control algorithm, we simulated a simple distributed system consisting of a single array and multiple hosts using Yacsim [17]. Each host runs an instance of the algorithm in a distributed manner, and the array services requests with latency based on an exponential distribution with a mean of $1/C$. We conducted a series of experiments with various capacities, workloads, and parameter values.

To test the algorithm's adaptability, we experimented with three hosts using a $1:2:3$ share ratio, $\mathscr{L} = 200$ ms, and an array capacity that changes from 400 req/s to 100 req/s halfway through the experiment. Figure 2 plots the throughput, window size and average latency observed by the hosts for a period of 200 seconds. As expected, the control algorithm drives the system to operate close to the desired



(a) Window size (b) Average Latency (c) Throughput

Figure 2: Simulation of three hosts with $1:2:3$ share ratio. Array capacity is reduced from 400 to 100 req/s at $t = 100$ s.

latency threshold $\mathscr{L}$. We also used the simulator to verify that as $\mathscr{L}$ is varied (100 ms, 200 ms and 300 ms), the system latencies operate close to $\mathscr{L}$, and that windows sizes increase while maintaining their proportional ratio.

3.3 End-to-End Support

PARDA flow control ensures that each host obtains a fair share of storage array capacity proportional to its IO shares. However, our ultimate goal for storage resource management is to provide control over service rates for the applications running in VMs on each host. We use a fair queuing mechanism based on SFQ [10] for our host-level scheduler. SFQ implements proportional-sharing of the host's issue queue, dividing it among VMs based on their IO shares when there is contention for the host-level queue.

Two key features of the local scheduler are worth noting. First, the scheduler doesn't strictly partition the host-level queue among VMs based on their shares, allowing them to consume additional slots that are left idle by other VMs which didn't consume their full allocation. This handles short-term fluctuations in the VM workloads, and provide some statistical multiplexing benefits. Second, the scheduler doesn't switch between VMs after every IO, instead scheduling a group of IOs per VM as long as they exhibit some spatial locality (within a few MB). These techniques have been shown to improve overall IO performance [3,13].

Combining a distributed flow control mechanism with a fair local scheduler allows us to provide end-to-end IO allocations to VMs. However, an interesting alternative is to apply PARDA flow control at the VM level, using per-VM latency measurements to control per-VM window sizes directly, independent of how VMs are mapped to hosts. This approach is appealing, but it also introduces new challenges that we are currently investigating. For example, per-VM allocations may be very small, requiring new techniques to support fractional window sizes, as well as efficient distributed methods to compensate for short-term burstiness.

3.4 Handling Bursts

A well-known characteristic of many IO workloads is a bursty arrival pattern—fluctuating resource demand due to device and application characteristics, access locality, and other factors. A high degree of burstiness makes it difficult to provide low latency and achieve proportionate allocation.

In our environment, bursty arrivals generally occur at two distinct time scales: systematic long-term ON-OFF behavior of VMs, and sudden short-term spikes in IO workloads. To handle long-term bursts, we modify the β value for a host based on the utilization of queue slots by its resident VMs. Recall that the host-level parameter β is proportional to the sum of shares of all VMs (if s_i are the shares assigned to VM i, then for host h, $\beta_h = K \times \sum_i s_i$, where K is a normalization constant).

To adjust β, we measure the average number of outstanding IOs per VM, n_k, and each VM's share of its host window size as w_k, expressed as:

$$w_k = \frac{s_k}{\sum_i s_i} w(t) \tag{4}$$

If $(n_k < w_k)$, we scale the shares of the VM to be $s'_i = n_k \times s_k / w_k$ and use this to calculate β for the host. Thus if a VM is not fully utilizing its window size, we reduce the β value of its host, so other VMs on the same host do not benefit disproportionately due to the under-utilized shares of a colocated idle VM. In general, when one or more VMs become idle, the control mechanism will allow all hosts (and thus all VMs) to proportionally increase their window sizes and exploit the spare capacity.

For short-term fluctuations, we use a burst-aware local scheduler. This scheduler allows VMs to accumulate a bounded number of credits while idle, and then schedule requests in bursts once the VM becomes active. This also improves overall IO efficiency, since requests from a single VM typically exhibit some locality. A number of schedulers support bursty allocations [6, 13, 22]. Our implementation uses SFQ as the local scheduler, but allows a bounded number of IOs to be batched from each VM instead of switching among VMs purely based on their SFQ request tags.

4 Storage-Specific Challenges

Storage devices are stateful and their throughput can be quite variable, making it challenging to apply the latency-based flow control approaches used in networks. Equilibrium may not be reached if different hosts observe very different latencies during overload. Next we discuss three key issues to highlight the differences between storage and network service times.

Request Location. It is well known that the latency of a request can vary from a fraction of a millisecond to tens of milliseconds, based on its location compared to previous requests, as well as caching policies at the array. Variability in seek and rotational delays can cause an order of magnitude difference in service times. This makes it difficult to estimate the baseline IO latency corresponding to the latency with no queuing delay. Thus a sudden change in average latency or in the ratio of current values to the previous average may or may not be a signal for overload. Instead, we look at average latency values in comparison to a latency threshold $\mathscr{L}$ to predict congestion. The assumption is that latencies observed during congestion will have a large *queuing delay* component, outweighing increases due to workload changes (*e.g.*, sequential to random).

Request Type. Write IOs are often returned to the host once the block is written in the controller's NVRAM. Later, they are flushed to disk during the destage process. However, read IOs may need to go to disk more often. Similarly, two requests from a single stream may have widely varying latencies if one hits in the cache and the other misses. In certain RAID systems [5], writes may take four times longer than reads due to parity reads and updates. In general, IOs from a single stream may have widely-varying response times, affecting the latency estimate. Fortunately, a moving average over a sufficiently long period can absorb such variations and provide a more consistent estimate.

IO Size. Typical storage IO sizes range from 512 bytes to 256 KB, or even 1 MB for more recent devices. The estimator needs to be aware of changing IO size in the workload. This can be done by computing latency per 8 KB instead of latency per IO using a linear model with certain fixed costs. Size variance is less of an issue in networks since most packets are broken into MTU-size chunks (typically 1500 bytes) before transmission.

All of these issues essentially boil down to the problem of estimating highly-variable latency and using it as an indicator of array overload. We may need to distinguish between latency changes caused by workload versus those due to the overload at the array. Some of the variation in IO latency can be absorbed by long-term averaging, and by considering latency per fixed IO size instead of per IO request. Also, a sufficiently high baseline latency (the desired operating point for the control algorithm, $\mathscr{L}$) will be insensitive to workload-based variations in under-utilized cases.

4.1 Distributed Implementation Issues

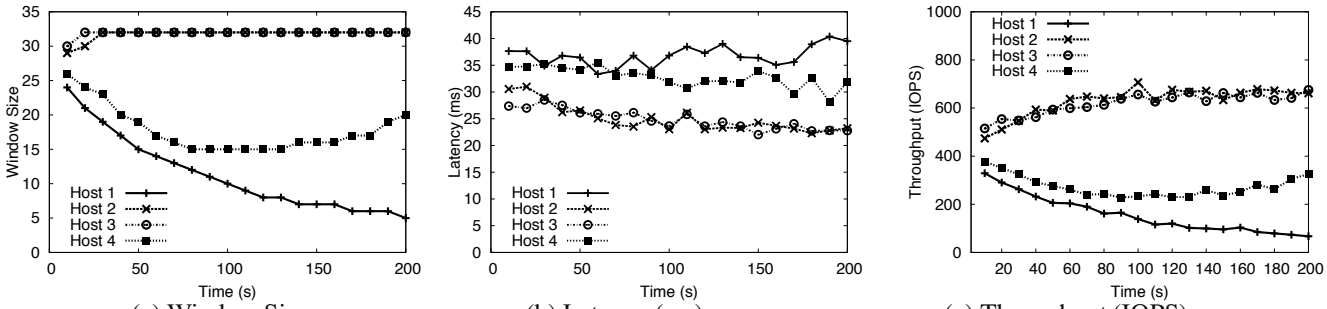
We initially implemented PARDA in a completely distributed manner, where each host monitored only its own IO latency to calculate $L(t)$ for Equation 2 (referred to as local latency estimation). However, despite the use of averaging, we found that latencies observed at different hosts were dependent on block-level placement.

We experimented with four hosts, each running one Windows Server 2003 VM configured with a 16 GB data disk created as a contiguous file on the shared LUN. Each VM also has a separate 4 GB system disk. The storage array was an EMC CLARiiON CX3-40 (same hardware setup as in Section 5). Each VM executed a 16 KB random read IO workload. Running without any control algorithm, we noticed that the hosts observed average latencies of 40.0, 34.5, 35.0 and 39.5 ms, respectively. Similarly, the throughput observed by the hosts were 780, 910, 920 and 800 IOPS respectively. Notice that hosts two and three achieved better IOPS and lower latency, even though all hosts were issuing exactly the same IO pattern.

We verified that this discrepancy is explained by placement: the VM disks (files) were created and placed in order on the underlying device/LUN, and the middle two virtual disks exhibited better performance compared to the two outer disks. We then ran the control algorithm with latency threshold $\mathscr{L}$ = 30 ms and equal β for all hosts. Figure 3 plots the computed window size, latency and throughput over a period of time. The discrepancy in latencies observed across hosts leads to divergence in the system. When hosts two and three observe latencies smaller than $\mathscr{L}$, they increase their window size, whereas the other two hosts still see latencies higher than $\mathscr{L}$, causing further window size decreases. This undesirable positive feedback loop leads to a persistent performance gap.

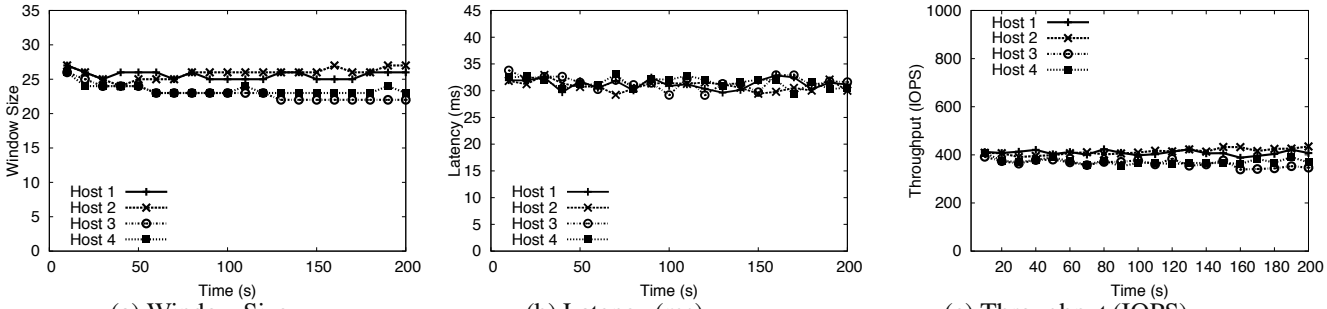
To validate that this effect is due to block placement of VM disks and array level scheduling, we repeated the same experiment using a single 60 GB shared disk. This disk file was opened by all VMs using a "multi-writer" mode. Without any control, all hosts observed a throughput of $\sim$ 790 IOPS and latency of 39 ms. Next we ran with PARDA on the shared disk, again using equal β and $\mathscr{L} = 30$ ms. Figure 4 shows that the window sizes of all hosts are reduced, and the cluster-wide latency stays close to 30 ms.

This led us to conclude that, at least for some disk subsystems, latency observations obtained individually at each host for its IOs are a fragile metric that can lead to divergences. To avoid this problem, we instead implemented a robust mechanism that generates a consistent signal for contention in the entire cluster, as discussed in the next section.



(a) Window Size (b) Latency (ms) (c) Throughput (IOPS)

Figure 3: Local L(t) Estimation. Separate VM disks cause window size divergence due to block placement and unfair array scheduling.



(a) Window Size (b) Latency (ms) (c) Throughput (IOPS)

Figure 4: Local L(t) Estimation. VMs use same shared disk, stabilizing window sizes and providing more uniform throughput and latency.

4.2 Latency Aggregation

After experimenting with completely decentralized approaches and encountering the divergence problem detailed above, we implemented a more centralized technique to compute cluster-wide latency as a consistent signal. The aggregation doesn't need to be very accurate, but it should be reasonably consistent across hosts. There are many ways to perform this aggregation, including approximations based on statistical sampling. We discuss two different techniques that we implemented for our prototype.

Network-Based Aggregation. Each host uses a UDP socket to listen for statistics advertised by other hosts. The statistics include the average latency and number of IOs per LUN. Each host either broadcasts its data on a common subnet, or sends it to every other host individually. This is an instance of the general average- and sum-aggregation problem for which multicast-based solutions also exist [29].

Filesystem-Based Aggregation. Since we are trying to control access to a shared filesystem volume (LUN), it is convenient to use the same medium to share the latency statistics among the hosts. We implement a shared file per volume, which can be accessed by multiple hosts simultaneously. Each host owns a single block in the file and periodically writes its average latency and number of IOs for the LUN into that block. Each host reads that file periodically using a single large IO and locally computes the cluster-wide average to use for window size estimation.

In our experiments, we have not observed extremely high variance across per-host latency values, although this seems possible if some workloads are served primarily from the storage array's cache. In any case, we do not anticipate that this would affect PARDA stability or convergence.

5 Experimental Evaluation

In this section, we present the results from a detailed evaluation of PARDA in a real system consisting of up to eight hosts accessing a shared storage array. Each host is a Dell Poweredge 2950 server with 2 Intel Xeon 3.0 GHz dual-core processors, 8 GB of RAM and two Qlogic HBAs connected to an EMC CLARiiON CX3-40 storage array over a Fibre Channel SAN. The storage volume is hosted on a 10-disk RAID-5 disk group on the array.

Each host runs the VMware ESX Server hypervisor [24] with a local instance of the distributed flow control algorithm. The aggregation of average latency uses the filesystem-based implementation described in Section 4.2, with a two-second update period. All PARDA experiments used the smoothing parameters $\alpha = 0.002$ and $\gamma = 0.8$.

Our evaluation consists of experiments that examine five key questions: (1) How does average latency vary with changes in workload? (2) How does average latency vary with load at the array? (3) Can the PARDA algorithm adjust issue queue lengths based on per-host latencies to provide differentiated service? (4) How well can this mechanism

handle bursts and idle hosts? (5) Can we provide end-to-end IO differentiation using distributed flow control together with a local scheduler at each host?

Our first two experiments determine whether average latency can be used as a reliable indicator to detect overload at the storage array, in the presence of widely-varying workloads. The third explores how effectively our control module can adjust host queue lengths to provide coarse-grained fairness. The remaining experiments examine how well PARDA can deal with realistic scenarios that include workload fluctuations and idling, to provide end-to-end fairness to VMs. Throughout this section, we will provide data using a variety of parameter settings to illustrate the adaptability and robustness of our algorithm.

5.1 Latency vs. Workload

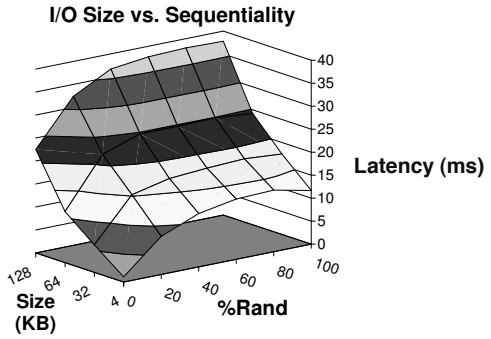


Figure 5: Latency as a function of IO size and sequentiality.

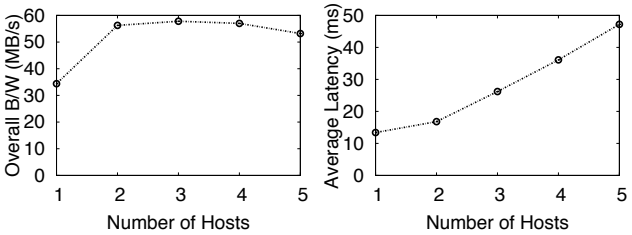
We first consider a single host running one VM executing different workloads, to examine the variation in average latency measured at the host. A Windows Server 2003 VM running *Iometer* [1] is used to generate each workload, configured to keep 8 IOs pending at all times.

We varied three workload parameters: reads – 0 to 100%, IO size – 4, 32, 64, and 128 KB, and sequentiality – 0 to 100%. For each combination, we measured throughput, bandwidth, and the average, min and max latencies.

Over all settings, the minimum latency was observed for the workload consisting of 100% sequential 4 KB reads, while the maximum occurred for 100% random 128 KB writes. Bandwidth varied from 8 MB/s to 177 MB/s. These results show that bandwidth and latency can vary by more than a factor of 20 due solely to workload variation.

Figure 5 plots the average latency (in ms) measured for a VM while varying IO size and degree of sequentiality. Due to space limitations, plots for other parameters have been omitted; additional results and details are available in [11].

There are two main observations: (1) the absolute latency value is not very high for any configuration, and (2) latency usually increases with IO size, but the slope is small because transfer time is usually dominated by seek and rotational delays. This suggests that array overload can be detected by using a fairly high latency threshold value.



(a) Aggregate Bandwidth (MB/s) (b) Average Latency (ms)

Figure 6: Overall bandwidth and latency observed by multiple hosts as the number of hosts is increased from 1 to 5.

Workload			Phase1			Phase2		
Size	Read	Random	Q	T	L	Q	T	L
16K	70%	60%	32	1160	26	16	640	24
16K	100%	100%	32	880	35	32	1190	27
8K	75%	0%	32	1280	25	16	890	17
8K	90%	100%	32	900	36	32	1240	26

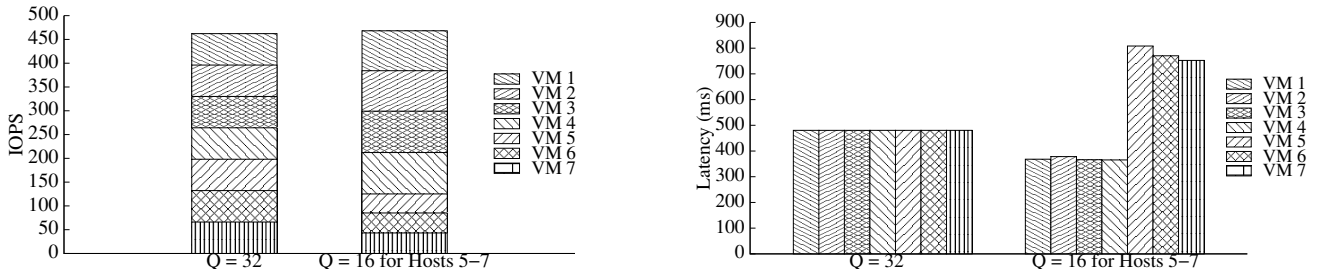
Table 2: Throughput (T IOPS) and latencies (L ms) observed by four hosts for different workloads and queue lengths (Q).

5.2 Latency vs. Queue Length

Next we examine how IO latency varies with increases in overall load (queue length) at the array. We experimented with one to five hosts accessing the same array. Each host generates a uniform workload of 16 KB IOs, 67% reads and 70% random, keeping 32 IOs outstanding. Figure 6 shows the aggregate throughput and average latency observed in the system, with increasing contention at the array. Throughput peaks at three hosts, but overall latency continues to increase with load. Ideally, we would like to operate at the lowest latency where bandwidth is high, in order to fully utilize the array without excessive queuing delay.

For uniform workloads, we also expect a good correlation between queue size and overall throughput. To verify this, we configured seven hosts to access a 400 GB volume on a 5-disk RAID-5 disk group. Each host runs one VM with an 8 GB virtual disk. We report data for a workload of 32 KB IOs with 67% reads, 70% random and 32 IOs pending. Figure 7 presents results for two different static host-level window size settings: (a) 32 for all hosts and (b) 16 for hosts 5, 6 and 7.

We observe that the VMs on the throttled hosts receive approximately half the throughput ($\sim$ 42 IOPS) compared to other hosts ($\sim$ 85 IOPS) and their latency ($\sim$ 780 ms) is doubled compared to others ($\sim$ 360 ms). Their reduced performance is a direct result of throttling, and the increased latency arises from the fact that a VM's IOs were queued at its host. The device latency measured at the hosts (as opposed to in the VM, which would include time spent in host queues) is similar for all hosts in both experiments. The



(a) Average IOPS (b) Average latency (ms)

Figure 7: VM bandwidth and latency observed when queue length $Q = 32$ for all hosts, and when $Q = 16$ for some hosts.

overall latency decreases when one or more hosts are throttled, since there is less load on the array. For example, in the second experiment, the overall average latency changes from $\sim$ 470 ms at each host to $\sim$ 375 ms at each host when the window size is 16 for hosts 5, 6, and 7.

We also experimented with four hosts sending different workloads to the array while we varied their queue lengths in two phases. Table 2 reports the workload description and corresponding throughput and latency values observed at the hosts. In phase 1, each host has a queue length of 32 while in phase 2, we lowered the queue length for two of the hosts to 16. This experiment demonstrates two important properties. First, overall throughput reduces roughly in proportion to queue length. Second, if a host is receiving higher throughput at some queue length Q due to its workload being treated preferentially, then even for a smaller queue length $Q/2$, the host still obtains preferential treatment from the array. This is desirable because overall efficiency is improved by giving higher throughput to request streams that are less expensive for the array to process.

5.3 PARDA Control Method

In this section, we evaluate PARDA by examining fairness, latency threshold effects, robustness with non-uniform workloads, and adaptation to capacity changes.

5.3.1 Fairness

We experimented with identical workloads accessing 16 GB virtual disks from four hosts with equal β values. This is similar to the setup that led to divergent behavior in Figure 3. Using our filesystem-based aggregation, PARDA converges as desired, even in the presence of different latency values observed by hosts. Table 3 presents results for this workload without any control, and with PARDA using equal shares for each host; plots are omitted due to space constraints. With PARDA, latencies drop, making the overall average close to the target $\mathscr{L}$. The aggregate throughput achieved by all hosts is similar with and without PARDA, exhibiting good work-conserving behavior. This demonstrates that the algorithm works correctly in the simple case of equal shares and uniform workloads.

	Uncontrolled		PARDA $\mathscr{L}$ = 30 ms		
Host	IOPS	Latency (ms)	β	IOPS	Latency (ms)
1	780	41	1	730	34
2	900	34	1	890	29
3	890	35	1	930	29
4	790	40	1	800	33
Aggregate	3360	Avg = 37		3350	Avg = 31

Table 3: Fairness with 16 KB random reads from four hosts.

Next, we experimented with a share ratio of 1 : 1 : 2 : 2 for four hosts, setting $\mathscr{L}$ = 25 ms, shown in Figure 8. PARDA converges on windows sizes for hosts 1 and 2 that are roughly half those for hosts 3 and 4, demonstrating good fairness. The algorithm also successfully converges latencies to $\mathscr{L}$. Finally, the per-host throughput levels achieved while running this uniform workload also roughly match the specified share ratio. The remaining differences are due to some hosts obtaining better throughput from the array, even with the same window size. This reflects the true IO costs as seen by the array scheduler; since PARDA operates on window sizes, it maintains high efficiency at the array.

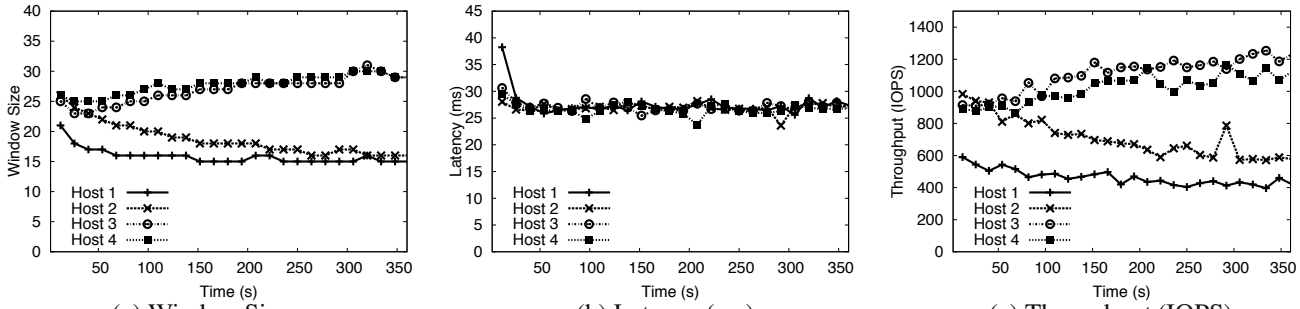
5.3.2 Effect of Latency Threshold

Recall that $\mathscr{L}$ is the desired latency value at which the array provides high throughput but small queuing delay. Since PARDA tries to operate close to $\mathscr{L}$, an administrator can control the overall latencies in a cluster, bounding IO times for latency-sensitive workloads such as OLTP. We investigated the effect of the threshold setting by running PARDA with different $\mathscr{L}$ values. Six hosts access the array concurrently, each running a VM with a 16 GB disk performing 16 KB random reads with 32 outstanding IOs.

Host	IOPS	Latency (ms)	Host	IOPS	Latency (ms)
1	525	59	4	560	57
2	570	55	5	430	77
3	570	55	6	500	62

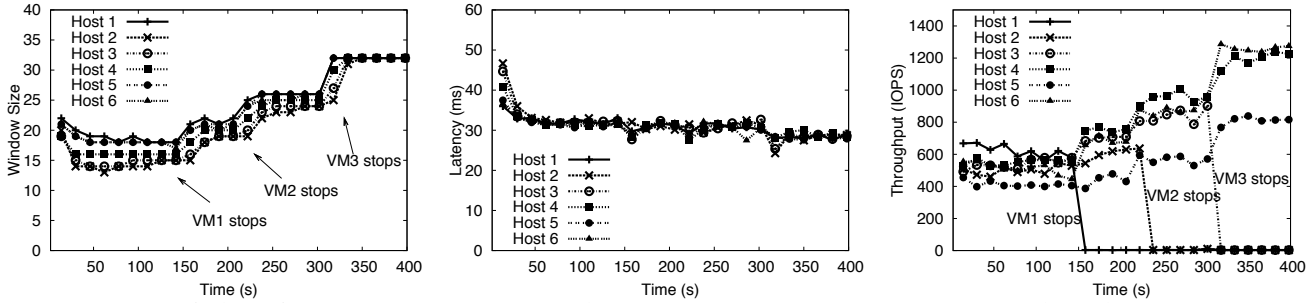
Table 4: Uncontrolled 16 KB random reads from six hosts.

We first examine the throughput and latency observed in the uncontrolled case, presented in Table 4. In Figure 9, we enable the control algorithm with $\mathscr{L}$ = 30 ms and equal shares, stopping one VM each at times t = 145 s, t = 220 s



(a) Window Size (b) Latency (ms) (c) Throughput (IOPS)

Figure 8: PARDA Fairness. Four hosts each run a 16 KB random read workload with β values of 1 : 1 : 2 : 2. Window sizes allocated by PARDA are in proportion to β values, and latency is close to the specified threshold $\mathscr{L}$ = 25 ms.



(a) Window Size (b) Latency (ms) (c) Throughput (IOPS)

Figure 9: PARDA Adaptation. Six hosts each run a 16 KB random read workload, with equal β values and $\mathscr{L}$ = 30 ms. VMs are stopped at t = 145 s, t = 220 s and t = 310 s, and window sizes adapt to reflect available capacity.

and t = 310 s. Comparing the results we can see the effect of the control algorithm on performance. Without PARDA, the system achieves a throughput of 3130 IOPS at an average latency of 60 ms. With $\mathscr{L}$ = 30 ms, the system achieves a throughput of 3150 IOPS, while operating close to the latency threshold. Other experiments with different threshold values, such as those shown in Figure 10 ($\mathscr{L}$ = 40 ms) and Figure 12 ($\mathscr{L}$ = 25 ms), confirm that PARDA is effective at maintaining latencies near $\mathscr{L}$.

These results demonstrate that PARDA is able to control latencies by throttling IO from hosts. Note the different window sizes at which hosts operate for different values of $\mathscr{L}$. Figure 9(a) also highlights the adaptation of window sizes, as more capacity becomes available at the array when VMs are turned off at various points in the experiment. The ability to detect capacity changes through changes in latency is an important dynamic property of the system.

5.3.3 Non-Uniform Workloads

To test PARDA and its robustness with mixed workloads, we ran very different workload patterns at the same time from our six hosts. Table 5 presents the uncontrolled case.

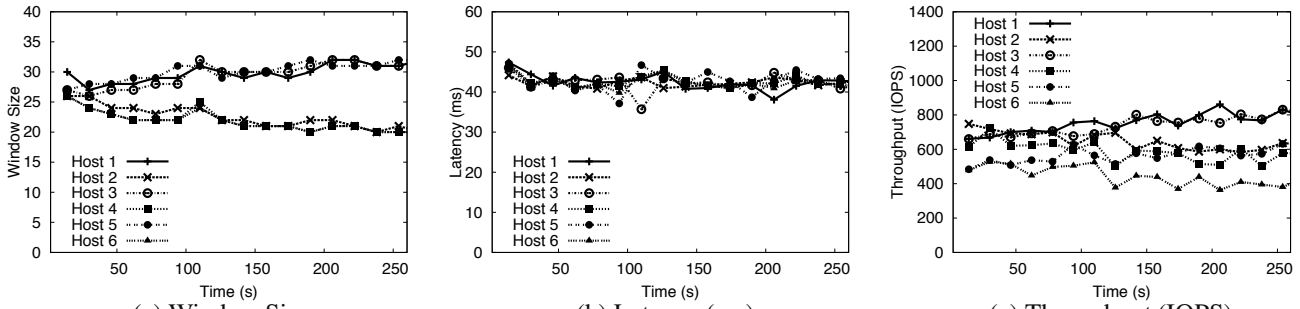
Next, we enable PARDA with $\mathscr{L}$ = 40 ms, and assign shares in a 2 : 1 : 2 : 1 : 2 : 1 ratio for hosts 1 through 6 respectively, plotted in Figure 10. Window sizes are differentiated between hosts with different shares. Hosts with more shares reach a window size of 32 (the upper bound, w_{max}) and remain there. Other hosts have window sizes close to 19. The average latency observed by the hosts remains close to $\mathscr{L}$, as shown in Figure 10(b). The throughput observed by hosts follows roughly the same pattern as window sizes, but is not always proportional because of array scheduling and block placement issues. We saw similar adaptation in window sizes and latency when we repeated this experiment using $\mathscr{L}$ = 30 ms (plots omitted due to space constraints).

Host	Size	Read	Random	IOPS	Latency (ms)
1	4K	100%	100%	610	51
2	8K	50%	0%	660	48
3	8K	100%	100%	630	50
4	8K	67%	60%	670	47
5	16K	100%	100%	490	65
6	16K	75%	70%	520	60

Table 5: Uncontrolled access by mixed workloads from six hosts.

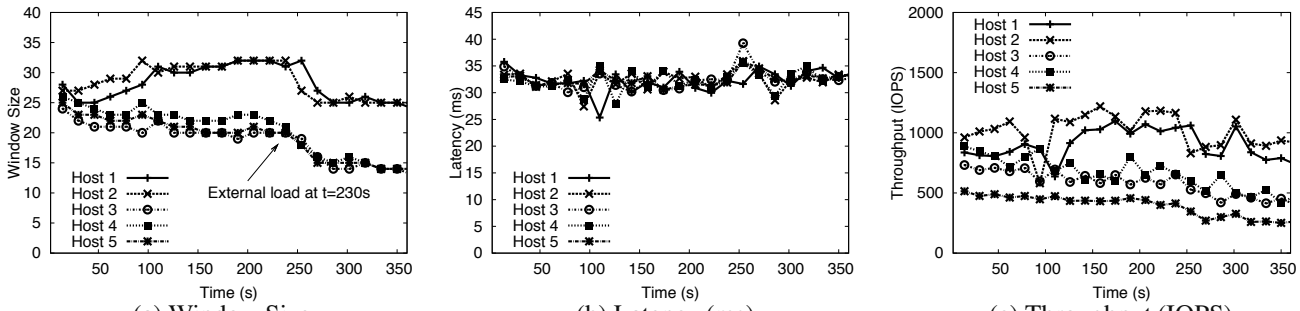
5.3.4 Capacity Changes

Storage capacity can change dramatically due to workload changes or array accesses by uncontrolled hosts external to PARDA. We have already demonstrated in Section 5.3.2 that our approach is able to absorb any spare capacity that becomes available. To test the ability of the control algorithm to handle decreases in capacity, we conducted an experiment starting with the first five hosts from the previous



(a) Window Size (b) Latency (ms) (c) Throughput (IOPS)

Figure 10: Non-Uniform Workloads. PARDA control with $\mathscr{L}$ = 40 ms. Six hosts run mixed workloads, with β values 2 : 1 : 2 : 1 : 2 : 1.



(a) Window Size (b) Latency (ms) (c) Throughput (IOPS)

Figure 11: Capacity Fluctuation. Uncontrolled external host added at t = 230 s. PARDA-controlled hosts converge to new window sizes.

experiment. At time t = 230 s, we introduce a sixth host that is not under PARDA control. This uncontrolled host runs a Windows Server 2003 VM issuing 16 KB random reads to a 16 GB virtual disk located on the same LUN as the others.

With $\mathscr{L}$ = 30 ms and a share ratio of 2 : 2 : 1 : 1 : 1 for the PARDA-managed hosts, Figure 11 plots the usual metrics over time. At t = 230 s, the uncontrolled external host starts, thereby reducing available capacity for the five controlled hosts. The results indicate that as capacity changes, the hosts under control adjust their window sizes in proportion to their shares, and observe latencies close to $\mathscr{L}$.

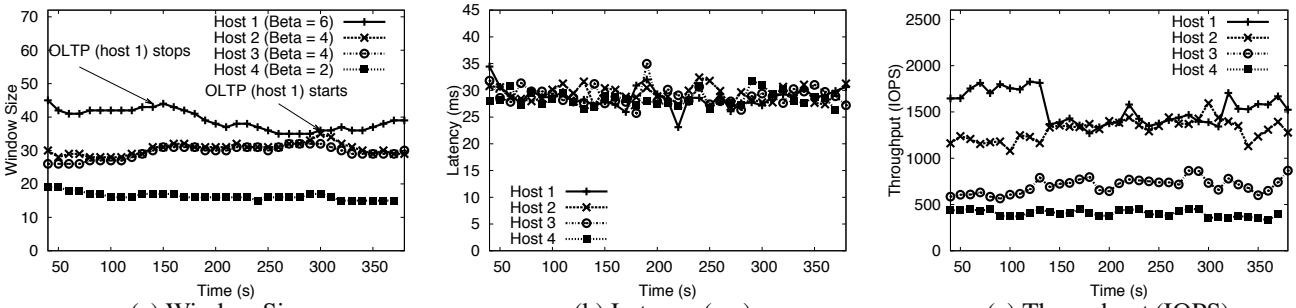
5.4 End-to-End Control

We now present an end-to-end test where multiple VMs run a mix of realistic workloads with different shares. We use Filebench [20], a well-known IO modeling tool, to generate an OLTP workload similar to TPC-C. We employ four VMs running Filebench, and two generating 16 KB random reads. A pair of Filebench VMs are placed on each of two hosts, whereas the micro-benchmark VMs occupy one host each. This is exactly the same experiment discussed in Section 2; data for the uncontrolled baseline case is presented in Table 1. Recall that without PARDA, hosts 1 and 2 obtain similar throughput even though the overall sum of their VM shares is different. Table 6 provides setup details and reports data using PARDA control. Results for the OLTP VMs are presented as Filebench operations per second (Ops/s).

Host	VM Type	s_1, s_2	β_h	$VM1$	$VM2$	T_h
1	2×OLTP	20, 10	6	1266 Ops/s	591 Ops/s	1857
2	2×OLTP	10, 10	4	681 Ops/s	673 Ops/s	1316
3	1×Micro	20	4	740 IOPS	n/a	740
4	1×Micro	10	2	400 IOPS	n/a	400

Table 6: PARDA end-to-end control for Filebench OLTP and micro-benchmark VMs issuing 16 KB random reads. Configured shares (s_i), host weights (β_h), Ops/s for Filebench VMs and IOPS (T_h for hosts) are respected across hosts. $\mathscr{L}$ = 25 ms, w_{max} = 64.

We run PARDA ($\mathscr{L}$ = 25 ms) with host weights (β_h) set according to shares of their VMs ($\beta_h = 6 : 4 : 4 : 2$ for hosts 1 to 4). The maximum window size w_{max} is 64 for all hosts. The OLTP VMs on host 1 receive 1266 and 591 Ops/s, matching their 2 : 1 share ratio. Similarly, OLTP VMs on host 2 obtain 681 and 673 Ops/s, close to their 1 : 1 share ratio. Note that the overall Ops/s for hosts 1 and 2 have a 3 : 2 ratio, which is not possible in an uncontrolled scenario. Figure 12 plots the window size, latency and throughput observed by the hosts. We note two key properties: (1) window sizes are in proportion to the overall β values and (2) each VM receives throughput in proportion to its shares. This shows that PARDA provides the strong property of enforcing VM shares, independent of their placement on hosts. The local SFQ scheduler divides host-level capacity across VMs in a fair manner, and together with PARDA, is able to provide effective end-to-end isolation among VMs. We also modified one VM workload during the experiment



(a) Window Size (b) Latency (ms) (c) Throughput (IOPS)

Figure 12: PARDA End-to-End Control. VM IOPS are proportional to shares. Host window sizes are proportional to overall β values.

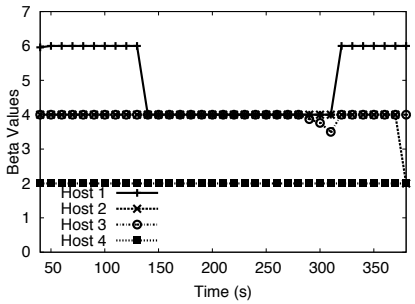


Figure 13: Handling Bursts. One OLTP workload on host 1 stops at t = 140 s and restarts at t = 310 s. The β of host 1 is adjusted and window sizes are recomputed using the new β value.

to test our burst-handling mechanism, which we discuss in the next section.

5.5 Handling Bursts

Earlier we showed that PARDA maintains high utilization of the array even when some hosts idle, by allowing other hosts to increase their window sizes. However, if one or more *VMs* become idle, the overall β of the host must be adjusted, so that backlogged VMs on the same host don't obtain an unfair share of the current capacity. Our implementation employs the technique described in Section 3.4.

We experimented with dynamically idling one of the OLTP VM workloads running on host 1 from the previous experiment presented in Figure 12. The VM workload is stopped at t = 140 s and resumed at t = 310 s. Figure 13 shows that the β value for host 1 adapts quickly to the change in the VM workload. Figure 12(a) shows that the window size begins to decrease according to the modified lower value of $\beta = 4$ starting from t = 140 s. By t = 300 s, window sizes have converged to a 1 : 2 ratio, in line with aggregate host shares. As the OLTP workload becomes active again, the dynamic increase in the β of host 1 causes its window size to grow. This demonstrates that PARDA ensures fairness even in the presence of non-backlogged workloads, a highly-desirable property for shared storage access.

		Uncontrolled			PARDA		
Host	VM Type	*OPM*	*Avg Lat*	T_h, L_h	β_h	*OPM*	*Avg Lat*
1	SQL1	8799	213	615, 20.4	1	6952	273
2	SQL2	8484	221	588, 20.5	4	12356	151

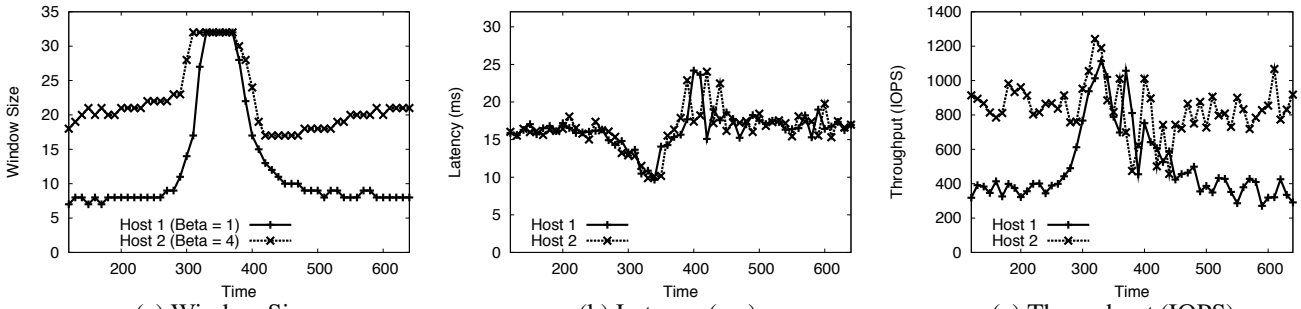
Table 7: Two SQL Server VMs with 1 : 4 share ratio, running with and without PARDA. Host weights (β_h) and OPM (orders/min), IOPS (T_h for hosts) and latencies (*Avg Lat* for database operations, L_h for hosts, in ms). $\mathscr{L}$ = 15 ms, w_{max} = 32.

5.6 Enterprise Workloads

To test PARDA with more realistic enterprise workloads, we experimented with two Windows Server 2003 VMs, each running a Microsoft SQL Server 2005 Enterprise Edition database. Each VM is configured with 4 virtual CPUs, 6.4 GB of RAM, a 10 GB system disk, a 250 GB database disk, and a 50 GB log disk. The database virtual disks are hosted on an 800 GB RAID-0 LUN with 6 disks; log virtual disks are placed on a 100 GB RAID-0 LUN with 10 disks. We used the Dell DVD store (DS2) database test suite, which implements a complete online e-commerce application, to stress the SQL databases [7]. We configured a 15 ms latency threshold, and ran one VM per host, assigning shares in a 1 : 4 ratio.

Table 7 reports the parameters and the overall application performance for the two SQL Server VMs. Without PARDA, both VMs have similar performance in terms of both orders per minute (OPM) and average latency. When running with PARDA, the VM with higher shares obtains roughly twice the OPM throughput and half the average latency. The ratio isn't 1 : 4 because the workloads are not always backlogged, and the VM with higher shares can't keep its window completely full.

Figure 14 plots the window size, latency and throughput observed by the hosts. As the overall latency decreases, PARDA is able to assign high window sizes to both hosts. When latency increases, the window sizes converge to be approximately proportional to the β values. Figure 15 shows the β values for the hosts while the workload is running, and highlights the fact that the SQL Server VM on host 2 cannot always maintain enough pending IOs to fill



(a) Window Size (b) Latency (ms) (c) Throughput (IOPS)

Figure 14: Enterprise Workload. Host window sizes and IOPS for SQL Server VMs are proportional to their overall β values whenever the array resources are contended. Between $t = 300$ s and $t = 380$ s, hosts get larger window sizes since the array is not contended.

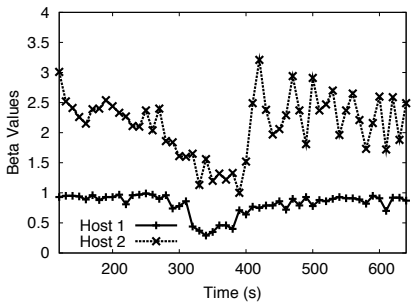


Figure 15: Dynamic β Adjustment. β values for hosts running SQL Server VMs fluctuate as pending IO counts change.

its window. This causes the other VM on host 1 to pick up the slack and benefit from increased IO throughput.

6 Related Work

The research literature contains a large body of work related to providing quality of service in both networks and storage systems, stretching over several decades. Numerous algorithms for network QoS have been proposed, including many variants of fair queuing [2, 8, 10]. However, these approaches are suitable only in centralized settings where a single controller manages all requests for resources. Stoica proposed QoS mechanisms based on a stateless core [23], where only edge routers need to maintain per-flow state, but some minimal support is still required from core routers.

In the absence of such mechanisms, TCP has been serving us quite well for both flow control and congestion avoidance. Commonly-deployed TCP variants use per-flow information such as estimated round trip time and packet loss at each host to adapt per-flow window sizes to network conditions. Other proposed variants [9] require support from routers to provide congestion signals, inhibiting adoption.

FAST-TCP [15] provides a purely latency-based approach to improving TCP's throughput in high bandwidth-delay product networks. In this paper we adapt some of the techniques used by TCP and its variants to perform flow control in distributed storage systems. In so doing, we have addressed some of the challenges that make it non-trivial to employ TCP-like solutions for managing storage IO.

Many storage QoS schemes have also been proposed to provide differentiated service to workloads accessing a single disk or storage array [4, 13, 14, 16, 25, 30]. Unfortunately, these techniques are centralized, and generally require full control over all IO. Proportionate bandwidth allocation algorithms have also been developed for distributed storage systems [12, 26]. However, these mechanisms were designed for brick-based storage, and require each storage device to run an instance of the scheduling algorithm.

Deployments of virtualized systems typically have no control over storage array firmware, and don't use a central IO proxy. Most commercial storage arrays offer only limited, proprietary quality-of-service controls, and are treated as black boxes by the virtualization layer. Triage [18] is one control-theoretic approach that has been proposed for managing such systems. Triage periodically observes the utilization of the system and throttles hosts using bandwidth caps to achieve a specified share of available capacity. This technique may underutilize array resources, and relies on a central controller to gather statistics, compute an online system model, and re-assign bandwidth caps to hosts. Host-level changes must be communicated to the controller to handle bursty workloads. In contrast, PARDA only requires very light-weight aggregation and per-host measurement and control to provide fairness with high utilization.

Friendly VMs [31] propose cooperative fair sharing of CPU and memory in virtualized systems leveraging feedback-control models. Without relying on a centralized controller, each "friendly" VM adapts its own resource consumption based on congestion signals, such as the relative progress of its virtual time compared to elapsed real time, using TCP-like AIMD adaptation. PARDA applies similar ideas to distributed storage resource management.

7 Conclusions

In this paper, we studied the problem of providing coarse-grained fairness to multiple hosts sharing a single storage system in a distributed manner. We propose a novel software system, PARDA, which uses average latency as an indicator for array overload and adjusts per-host issue queue lengths in a decentralized manner using flow control.

Our evaluation of PARDA in a hypervisor shows that it is able to provide fair access to the array queue, control overall latency close to a threshold parameter and provide high throughput in most cases. Moreover, combined with a local scheduler, PARDA is able to provide end-to-end prioritization of VM IOs, even in presence of variable workloads.

As future work, we are trying to integrate soft limits and reservations to provide a complete IO management framework. We would also like to investigate applications of PARDA to other non-storage systems where resource management must be implemented in a distributed fashion.

Acknowledgements

Thanks to Tim Mann, Minwen Ji, Anne Holler, Neeraj Goyal, Narasimha Raghunandana and our shepherd Jiri Schindler for valuable discussions and feedback. Thanks also to Chethan Kumar for help with experimental setup.

References

[1] Iometer. http://www.iometer.org.

[2] BENNETT, J. C. R., AND ZHANG, H. WF^2Q: Worst-case fair weighted fair queueing. In *Proc. of IEEE INFOCOM '96* (March 1996), pp. 120–128.

[3] BRUNO, J., BRUSTOLONI, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. Disk scheduling with quality of service guarantees. In *Proc. of the IEEE Int'l Conf. on Multimedia Computing and Systems, Volume 2* (1999), IEEE Computer Society.

[4] CHAMBLISS, D. D., ALVAREZ, G. A., PANDEY, P., JADAV, D., XU, J., MENON, R., AND LEE, T. P. Performance virtualization for large-scale storage systems. In *Symposium on Reliable Distributed Systems* (October 2003), pp. 109–118.

[5] CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys 26*, 2 (1994).

[6] CRUZ, R. L. Quality of service guarantees in virtual circuit switched networks. *IEEE Journal on Selected Areas in Communications 13*, 6 (1995), 1048–1056.

[7] DELL, INC. DVD Store. http://delltechcenter.com/page/DVD+store.

[8] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queuing algorithm. *Journal of Internetworking Research and Experience 1*, 1 (September 1990), 3–26.

[9] FLOYD, S., AND JACOBSON, V. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking 1*, 4 (1993), 397–413.

[10] GOYAL, P., VIN, H. M., AND CHENG, H. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking 5*, 5 (1997).

[11] GULATI, A., AND AHMAD, I. Towards distributed storage resource management using flow control. In *Proc. of First International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability* (2008).

[12] GULATI, A., MERCHANT, A., AND VARMAN, P. *d*Clock: Distributed QoS in heterogeneous resource environments. In *Proc. of ACM PODC (short paper)* (August 2007).

[13] GULATI, A., MERCHANT, A., AND VARMAN, P. *p*Clock: An arrival curve based approach for QoS in shared storage systems. In *Proc. of ACM SIGMETRICS* (June 2007), pp. 13–24.

[14] HUANG, L., PENG, G., AND CHIUEH, T. Multi-dimensional storage virtualization. In *Proc. of ACM SIGMETRICS* (June 2004).

[15] JIN, C., WEI, D., AND LOW, S. FAST TCP: Motivation, Architecture, Algorithms, Performance. *Proceedings of IEEE INFOCOM '04* (March 2004).

[16] JIN, W., CHASE, J. S., AND KAUR, J. Interposed proportional sharing for a storage service utility. In *ACM SIGMETRICS* (June 2004).

[17] JUMP, J. R. Yacsim reference manual. http://www.owlnet.rice.edu/~elec428/yacsim/yacsim.man.ps.

[18] KARLSSON, M., KARAMANOLIS, C., AND ZHU, X. Triage: Performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage 1*, 4 (2005), 457–480.

[19] LUMB, C., MERCHANT, A., AND ALVAREZ, G. Façade: Virtual storage devices with performance guarantees. *Proc. of File and Storage Technologies (FAST)* (March 2003).

[20] MCDOUGALL, R. Filebench: A prototype model based workload for file systems, work in progress. http://solarisinternals.com/si/tools/filebench/filebench_nasconf.pdf.

[21] POVZNER, A., KALDEWEY, T., BRANDT, S., GOLDING, R., WONG, T. M., AND MALTZAHN, C. Efficient guaranteed disk request scheduling with Fahrrad. *SIGOPS Oper. Syst. Rev. 42*, 4 (2008), 13–25.

[22] SARIOWAN, H., CRUZ, R. L., AND POLYZOS, G. C. Scheduling for quality of service guarantees via service curves. In *Proceedings of the International Conference on Computer Communications and Networks* (1995), pp. 512–520.

[23] STOICA, I., SHENKER, S., AND ZHANG, H. Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high speed networks. *IEEE/ACM Transactions on Networking 11*, 1 (2003), 33–46.

[24] VMWARE, INC. *Introduction to VMware Infrastructure*. 2007. http://www.vmware.com/support/pubs/.

[25] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: performance insulation for shared storage servers. In *Proc. of File and Storage Technologies (FAST)* (Feb 2007).

[26] WANG, Y., AND MERCHANT, A. Proportional-share scheduling for distributed storage systems. In *Proc. of File and Storage Technologies (FAST)* (Feb 2007).

[27] WONG, T. M., GOLDING, R. A., LIN, C., AND BECKER-SZENDY, R. A. Zygaria: Storage performance as a managed resource. In *Proc. of Real-Time and Embedded Technology and Applications Symposium* (April 2006), pp. 125–34.

[28] WU, J. C., AND BRANDT, S. A. The design and implementation of Aqua: an adaptive quality of service aware object-based storage device. In *Proc. of MSST* (May 2006), pp. 209–18.

[29] YALAGANDULA, P. A scalable distributed information management system. In *Proc. of SIGCOMM* (2004), pp. 379–390.

[30] ZHANG, J., SIVASUBRAMANIAM, A., WANG, Q., RISKA, A., AND RIEDEL, E. Storage performance virtualization via throughput and latency control. In *Proc. of MASCOTS* (September 2005).

[31] ZHANG, Y., BESTAVROS, A., GUIRGUIS, M., MATTA, I., AND WEST, R. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In *Proc. of Intl. Conference on Virtual Execution Environments (VEE)* (June 2005).