Improving storage bandwidth guarantees with performance insulation

Matthew Wachs, Gregory R. Ganger

CMU-PDL-10-113
October 2010

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Workloads that share a storage system should achieve predictable, controllable performance despite the activities of other workloads. One desirable way of expressing performance goals is as bandwidth guarantees. Unfortunately, storage bandwidth is difficult to allocate and manage among workloads, because total system capacity depends on both the workloads' access patterns and on any interference between them. This report demonstrates a new approach to supporting soft bandwidth guarantees, building on explicit performance insulation that bounds interference among workloads and its effect on performance and total system capacity. Combining dynamic disk head timeslicing and slack assignment, this approach eliminates almost all avoidable guarantee violations, leaving just those fundamental ones faced by individual workloads whose locality change too significantly. Experiments with a prototype show an order-of-magnitude decrease in the number of guarantee violations compared to traditional token-bucket based throttling.

Acknowledgements: We thank the members and companies of the PDL Consortium (including APC, EMC, Facebook, Google, HP, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Riverbed, Samsung, Seagate, STEC, Symantec, VMware, and Yahoo!) for their interest, insights, feedback, and support. We also thank Intel and NetApp for hardware donations that enabled this work. This material is based on research sponsored in part by the National Science Foundation, via grants #CNS-0326453, #CCF-0621499, and #CNS-0917012, by the Department of Energy, under award number DE-FC02-06ER25767, by the Army Research Office, under agreement number DAAD19–02–1–0389, and by an NDSEG Fellowship sponsored by the Department of Defense.

Keywords: QoS, quality of service, performance guarantees, storage performance, virtualization, performance insulation, performance isolation, shared storage

1 Introduction

Consolidation and physical resource sharing are core tenets of modern efforts to improve data center efficiency — resource utilization increases as different workloads[1] often vary in intensity independently and can be multiplexed on fewer resources than would be needed to accommodate each's individual surges. But, with such consolidation comes the potential for interference among workloads. Unmanaged, such interference can lead to unpredictable and highly variable performance, making administration difficult. In the case of storage, it can also lead to significant inefficiency due to disrupted locality. Both can reduce the benefits of consolidation.

An ideal sought for shared infrastructures is that of guaranteed service level objectives (SLOs) specified for each workload, sometimes called "storage QoS" for shared storage. Guided by SLOs, a shared infrastructure should reserve and schedule resources to provide predictable and controllable performance to each of its workloads, independent of the activities of other workloads.

For storage, SLO goals are commonly specified in terms of I/O bandwidth or latency percentiles (e.g., 90% of I/O requests completed within a certain response time). Such SLO metrics create an interesting challenge, because the fraction of a disk needed to provide a particular bandwidth or response time varies dramatically, depending on I/O locality in the workload. A modern disk might provide over 100 MB/s for sequential access, but less than 1 MB/s for random 4 KB requests. Moreover, mixing the I/O of multiple workloads can disrupt locality for each, potentially making the resources required a complex function of each workload's individual I/O locality and how their collective I/O requests are interleaved during servicing.

This report describes a new approach to providing for SLOs specified in terms of storage bandwidth. Unlike previous approaches, which largely ignore the effects of locality interference, our approach uses explicit performance insulation (most notably, via disk head timeslicing) to bound the effects of such interference. Entire disk head timeslices, rather than individual I/O requests, are interleaved, preventing one workload's variations from inducing uncontrolled or unpredictable changes in another's efficiency. Further, timeslices are dynamically resized, in each round, to mitigate the effects of locality variation within individual workloads.

Experiments with a prototype confirm the challenges of workload interference, the shortcomings of previous approaches in coping with it, and the success of our new approach. As expected, with multiple workloads, doing nothing results in frequent violations both due to a workload's locality changing significantly (fundamental) and due to unmanaged interference between workloads (avoidable). Perhaps surprisingly, the most popular traditional approach, based on throttling workloads with some form of token-bucket scheme, increases the frequency of avoidable violations (but reduces the worst magnitudes). Our approach removes almost all avoidable violations and also eliminates most of the ones that are "fundamental," given hard timeslicing. The latter is enabled by explicit assignment of slack to the workloads that need it most in any round, statistically exploiting the independent variation among workloads — one may need less in the same round as another needs more. The overall result is an order-of-magnitude reduction in the number of guarantee violations for the challenging workload mixes studied.

The remainder of this report is organized as follows. Section 2 provides additional background and discusses related work. Section 3 describes concepts underlying our scheduler's design. Section 4 details our algorithms and their implementation. Section 5 evaluates our approach, including comparison to prior schemes.

[1] We will use "workload" to denote a tenant of a shared infrastructure, such as an application or service.

2 Background and related work

When applications are "moved into the cloud," whether a "public cloud" or a "private cloud" (e.g., a VM-based cluster), they should not be made to suffer significant performance loss compared to their dedicated-infrastructure levels. Shared infrastructures could prevent this deterioration by supporting service level objectives. Each workload's CPU, memory, network, and storage demand can be expressed as SLOs, which are then set up in a management tool. A control system can allocate and manage each of these resources appropriately across the workloads to provide the desired levels of service.

Service level objectives for storage workloads are often expressed as bandwidth requirements, such as "Workload B needs 6 MB/s." Thus, a controller must be provided that is able to accept and achieve guarantees in this form. Of course, the core performance-sensitive workload behavior may not be the only activity that requires access to the workload's storage. The workload may include periods of other storage requests, such as those coming from maintenance tasks, for which the umbrella SLO does not make sense and is neither expected nor required. If the behavior of these non-core tasks is markedly different than the primary task, then the guarantee may not even be achievable (due to disk performance limitations) for the unusual requests.

To effectively use shared storage, each workload that has performance requirements must receive predictable and controllable performance. In addition, this performance should be achieved efficiently. Because workloads may have periods of activity during which their general performance goals are not practical or meaningful, the behavior of the system during these periods should be well-understood, fair to the other workloads, and reasonable.

2.1 Efficiency and performance insulation

Temporal and spatial locality are important factors in the performance of workloads when they access a hard disk. Because hard drives are mechanical devices, positioning times are significant. Incurring unnecessary repositioning delays can reduce the efficiency with which a workload is handled by an order of magnitude or more. When multiple workloads access the same disk concurrently, the combined request stream may contain interleaved requests from the different workloads. For many combinations of workloads, the locality of this combined stream is significantly lower than the locality within the individual workloads' streams. As a result, the disk head may need to frequently seek back and forth between the workloads' respective files, a potentially costly operation. While disk firmware is locality-aware and can reorder a sequence of requests in the queue to reduce seek times, queue depths at the firmware level can be far too short to provide high efficiency levels for concurrent workloads. For instance, for concurrent streaming workloads (ones with sequential block access patterns), efficiency may be below 30%.

Interference levels can vary dramatically — by an order of magnitude or more — over time. Even if Workload *A* has highly regular behavior and would receive stable performance on a dedicated system, the effects of Workload *B* can cause unexpected efficiency swings. This occurs when the intensity or access patterns of *B* change over time, resulting in periods when the combined request stream is especially unfavorable. Systems that do not manage interference allow the variance in one workload to cause unpredictability for all workloads. If Workload *A* suffers guarantee violations as a result of this variance, such violations should be considered avoidable, because they were caused by the failure of the sharing policy to maintain acceptable insulation from *B*. This type of violation is distinct from one caused by a workload's guarantee fundamentally being unsupportable for its access pattern when limited to its assigned resources.

Explicit timeslicing of disk head time is one approach for maintaining the intra-workload efficiency levels of workloads as they share a disk. The Argon storage server [16] (on which this work is based) uses long timeslices, e.g. 140 ms, to provide each workload long spans of uninterrupted access to the disk. In the middle of a workload's timeslice, the workload should receive the same efficiency as it would receive on a dedicated disk. Inefficiency comes when transitioning between timeslices, because the disk must seek from one workload's files to another's. By using timeslices that are significantly longer than the disk's worst-case seek time, Argon is able to amortize the cost of switching between workloads against a large span of "useful work."

But, Argon does not provide bandwidth or throughput guarantees in absolute terms, leaving unclear how to use it with such SLOs. Instead, workloads are assigned a fraction of disk time, e.g. 1/3, and the system is assigned a minimum efficiency level (called the *R-value*, e.g. 0.9 or 90%). By managing interference at the disk and cache, Argon then provides a bandwidth guarantee to a given workload A that is a function of the workload's dedicated-disk bandwidth, $standalone_A$. For instance, for the example $fraction_A$ and system efficiency level given above, Workload A would receive $1/3 \times 0.9 \times standalone_A$. The other workloads $B, C, \ldots$ in the system would receive similar levels of bandwidth corresponding to their respective values of $fraction_{B,C,\ldots}$ and $standalone_{B,C,\ldots}$.

One issue with this style of guarantee is that $standalone$ may vary as the access pattern of a workload changes. For example, at one point in time the workload may have a sequential access pattern, and $standalone$ may be 60 MB/s. Later, the workload may have a random access pattern, and $standalone$ may be 3 MB/s. If the workload has been assigned 1/3 of disk time and the system is operating at an efficiency level of 0.9, then when the workload is in the first operating region, its bandwidth will be at least 18 MB/s; but when the workload is in the second operating region, its bandwidth will only be guaranteed to be at least 0.9 MB/s. For workloads needing consistent bandwidth levels, choosing appropriate fractions and efficiency levels adds administrative complexity. Having to select a constant fraction for each workload is also a significant constraint, because the appropriate fraction may vary over time. For these reasons, a system that is able to discover the correct value of $fraction$ for each workload, and that is able to vary it over time, would be a desirable improvement.

2.2 Absolute bandwidth guarantees

There are two common approaches to providing bandwidth guarantees: bandwidth-based request throttling and deadline-based scheduling. Unfortunately, systems implementing these approaches have not fully addressed efficiency, resulting either in guarantee violations that would have been avoidable in a more efficient system, or the rejection of workload-guarantee combinations that would have been supportable in an efficient system.

2.2.1 Bandwidth-based throttling

Bandwidth-based request throttling systems generally manage workloads and requests using a *token-bucket model* [15]. Such a model establishes a rate (bandwidth) for each workload, and also permits a bounded amount of burstiness for each workload so long as they remain close to their specified rates. A system following this model allows a workload to issue requests unless it would exceed its assigned rate. Once a workload has received its assigned bandwidth in a period of time, additional requests are queued until the workload's average drops below its bandwidth limit.

Token-bucket systems suffer from a core mismatch between guarantees and limits. The express goal of a bandwidth guarantee is to provide a lower bound on bandwidth. However, by its very nature, a token-bucket system limits workloads to a rate, in other words, an upper bound. This dissonance is justified by the following argument: Suppose all the workloads and their corresponding guarantees *are* able to fit together on a particular system. We do not necessarily know, however, if the workloads will still fit if one or more is allowed to exceed its guarantee. Thus, to be cautious, we prevent any workload from receiving more than it needs. There are three problems with this argument. First, it is not always known *a priori* whether the workloads can, in fact, fit together at all times. If they cannot, then limiting workloads will not make them fit, and the "guarantees" provided by the system are violated anyway. Second, limiting workloads to exactly their minimum may be overly cautious; but it is not always clear when it is safe to provide additional disk time and when it is not. Third, the workloads might not fit on a system with a simple token-bucket scheduler, but may have been able to fit perfectly on the same system if it were using a more efficient scheduling policy. Because efficiency is not managed, a workload may receive its guarantee at one point in time but not at another, even though its behavior may not have changed; varying interference levels caused by other workloads may cause such unpredictability.

A number of systems use a token-bucket design, with various modifications to the general policy. SLEDS [3] allows workloads to exceed the strict token-bucket limit if no workload is exceeding its latency goal. Triage [9] performs token-based throttling, but also uses feedback to manage queue depths based on a tradeoff between latency (shorter queues) and potentially better disk scheduling (longer queues). While this may improve efficiency over simple token-bucket systems, total system throughput remains unpredictable. This led the authors to propose a novel form of guarantees that are parameterized by the level of total throughput the system is currently providing (which varies, in part, with efficiency). AVATAR and SARC [18] are a hierarchical pair of token-bucket and deadline-based schedulers. *p*Clock [4] also combines deadlines and a token-based model. Zygaria [17] improves the efficiency of sequential I/Os to a limited extent by treating sequential *runs* of requests, up to a fixed maximum size, as a single request in their token-bucket system. While some of these systems may improve efficiency beyond a basic token-bucket scheduler, none guarantees any minimum level.

2.2.2 Deadline-based scheduling

Deadline-based systems manage a queue of pending requests, each labeled with the time by which it must be issued or completed in order to meet its latency target. (For closed systems, bandwidth guarantees can be converted into latency requirements using a form of Little's law [10].) The system can sort requests by deadline and either "sneak ahead" requests belonging to background tasks in the slack before the earliest request must complete, or perform a limited amount of request reordering for better seek scheduling in this slack time. For workloads that need strong maximum latency bounds, deadline-based schedulers may provide the best possible performance. For all other types of workloads, however, they suffer from significant limitations. First, the relatively short queue lengths over which they can perform request reordering does not permit them to consistently achieve high levels of efficiency. Second, the efficiency they provide is highly variable and dependent on the overall combination of workloads and guarantees in a complex manner. Thus, like token-bucket schedulers, it is not clear whether a particular guarantee can be accommodated, and whether changing behavior from other workloads will lead to violations later even if guarantee adherence is currently perfect.

Various systems use variants of deadline-based scheduling. *p*Clock [4] and AVATAR/SARC [18], mentioned earlier, use both tokens and deadlines. Like Triage, Façade [11] uses feedback-based control to shorten queues when latency is not acceptable, and to lengthen queues when able in the hopes of improved disk scheduling. Stonehenge [6] "sneaks requests ahead" of upcoming deadlines when there is slack, dispatching these opportunistic requests in CSCAN order for possible efficiency improvements. RT-FS [12] also exploits slack to improve service beyond guarantees or provide best-effort allocations to non-guarantee workloads; in either case, SCAN order is used for better efficiency. Like the enhanced token-bucket schedulers, the improved efficiency levels provided by these schedulers are not guaranteed to be high.

2.3 Other related work

Other systems have been proposed for storage quality of service. Cello [14] differentiates between classes of workloads rather than individual workloads, and tries to provide a way to coordinate the potentiallyconflicting decisions each class's scheduler may make. Some systems focus on proportion rather than specific bandwidth or latency levels. YFQ [2] controls fairness but acknowledges that it may sacrifice throughput (i.e., efficiency) to meet this goal. Argon [16], our prior work described earlier, maintains high efficiency but offers guarantees in the form of fractions (utilization) and efficiency rather than bandwidth. Fahrrad [13] and the system proposed by Kaldewey *et al.* [8] also use utilization-based guarantees. Anticipatory scheduling [7] improves the efficiency of workloads by allowing them to maintain their locality even if they have brief periods of idleness that would otherwise lead to fine-grained interleaving of another workload's non-local requests. Argon's timeslicing avoids the same scenario.

3 Storage QoS scheduler design

In contrast to previous bandwidth guarantee systems, which largely ignore the effect of inter-workload interference on efficiency, we wish to design a scheduler that provides bandwidth guarantees while maintaining efficiency. With better efficiency, we expect better guarantee adherence. This section describes the concepts contributing to our scheduler, which itself is detailed in the next section.

3.1 Maintaining efficiency

To maintain efficiency, we utilize disk head timeslicing. As discussed earlier, the Argon storage server demonstrated that efficiency can be maintained when workloads are sharing a storage system by timeslicing the disk head's time across the workloads. For streaming workloads (workloads that read or write blocks sequentially from the disk), high efficiency (e.g., $> 80\%$) cannot be provided unless they are able to access the disk for long periods of time without having their spatial locality interrupted by any other workload. The efficiency level achieved for these workloads, in fact, can be directly computed using the amount of uninterrupted time they are assigned and the cost of "context switching" (seeking the disk head to the appropriate position within a streaming workload's file) at the beginning of their access periods. If a disk seek takes 10 ms and 90% efficiency is desired, for instance, then such a workload must be provided uninterrupted access for 90 ms at a time. Thus, we believe that a scheduler must incorporate some degree of timeslicing in order to be efficient, and choose to build on Argon's timeslicing in this work.

3.2 Providing bandwidth guarantees

However, in contrast to the goal of the Argon storage server, we wish to provide bandwidth guarantees. Thus, instead of sizing timeslices to apportion total disk time fractionally across workloads, our modified timeslice scheduler will size timeslices to meet bandwidth guarantees. In particular, let a *round* be the period of time during which each workload's timeslice executes once; our timeslicing-based scheduler will execute round after round indefinitely. We choose an appropriate round length (in milliseconds) that will be divided among the workloads. Then, for each workload, we estimate what fraction of the round it will need to achieve its bandwidth guarantee over that period. For instance, if the round is 2000 ms long, and a particular workload is assigned a guarantee of 1 MB/s, then it will be assigned a timeslice that is predicted to be long enough for it to transfer 2 MB. Since its timeslices are scheduled once every 2 seconds, this will yield an average bandwidth of 1 MB/s.

3.3 Handling different access patterns

Unfortunately, estimating appropriate timeslice lengths is not straightforward. For a streaming workload, it may take 33 ms to transfer 2 MB, while for a random workload, it may take 500–1000 ms. Other factors also affect performance, such as cache hit rate; and while sequential and random access patterns represent extreme examples, there is a continuum of spatial locality in between. Making matters worse, workloads may exhibit constant change in access patterns; what seemed to be the proper timeslice length at the beginning of a timeslice may prove to be highly inaccurate.

However, we can monitor the performance a workload is receiving even while its timeslice is progressing and make adjustments to our scheduling decisions proactively rather than reactively. In particular, if it becomes clear that a workload will finish enough requests to satisfy its bandwidth guarantee for a round much earlier than expected, we can terminate the timeslice before it was initially scheduled to complete. Similarly, if a workload is performing worse than expected, we may be able to extend its timeslice beyond the original target. Of course, doing so may come at the cost of shortening later timeslices and cause guarantee violations for those workloads. But, if a previous workload's timeslice was shortened because its guarantee has already been satisfied, this creates slack that could be assigned to later workloads without compromising any guarantees — and, in fact, the extra time may salvage a later workload's chances of meeting its guarantee for the round.

3.4 Fundamental vs. avoidable guarantee violations

Hence, there is a delicate balance between workloads that must be managed properly. Our goal is to honor the guarantees for each workload. When is it safe to assign more time to Workload *A* (decreasing its chances of a violation) at the expense of Workload *B* (increasing its chances of a violation)? To make the proper tradeoffs, we must consider if and when guarantee violations are "acceptable."

To address this question, we focus on an aspect of bandwidth guarantees that has not been adequately considered in prior systems: the issue of changing access patterns. Suppose a workload is sequential and is assigned a guarantee of 30 MB/s. Such a guarantee may present no challenge to the system. Later, however, the workload may become significantly more random; so much so that 30 MB/s may no longer be possible, even with full use of a dedicated disk. (Many disks cannot provide above 10 MB/s for random workloads.) This would result in a bandwidth guarantee violation no matter how sophisticated the bandwidth guarantee algorithm.

To prevent this scenario, some prior work [17] assumes each workload has a worst-case random access pattern when making admission decisions. Thus, the 30 MB/s guarantee would never be accepted in the first place, even if the workload never wavers from sequentiality and could easily achieve that level of performance. Thus, the problem of changing workloads is avoided by extreme conservatism, but the set of workloads that can be accommodated is also extremely conservative (i.e., much smaller than necessary). Most other systems leave this issue unaddressed; if the workload becomes unsupportable because of a change in access pattern, both it and the other workloads may begin to suffer guarantee violations. It will suffer, because it cannot be accommodated with the available resources; and other workloads will suffer, because in futilely trying to provide it with its performance requirement, resources will be directed away from other, supportable workloads. Note that "punishing" or terminating the guarantee for the unsupportable workload is not necessarily viable, because there is no notion of whether a particular access pattern is reasonable or unreasonable.

We address this problem by incorporating into our bandwidth guarantees the notion of *maximum fraction*. A workload is assigned both a bandwidth guarantee, such as 6 MB/s, and a maximum fraction, such as 1/3. Our system maintains the bandwidth guarantee so long as doing so does not result in the workload consuming more than the specified fraction of server time. In addition, if there is additional time available once other workloads are satisfied, a workload that has reached its resource limit without meeting its guarantee may receive additional *slack* beyond its maximum fraction, and may thus salvage its bandwidth guarantee. But, the system makes no promises that slack will be available.

This style of guarantee allows us to divide the set of guarantee violations into two classes: *fundamental* and *avoidable*. Fundamental violations are guarantee violations that occur when a workload's bandwidth cannot be accommodated within the fraction of time assumed when it was admitted; this occurs when the workload's access pattern has too little locality to achieve its bandwidth within its assigned resource fraction. Avoidable violations are the remaining violations, which are caused by failures of the sharing policy. Avoidable violations can occur because the sharing policy does not manage the set of workloads properly (for instance, it fails to maintain fairness among the workloads) or because the sharing policy does not maintain efficiency (it allows one workload's behavior to effect the efficiency another workload receives from the disk). These violations can be regarded as "artificial" sources of guarantee violations.

Because fundamental violations are a characteristic of the workload and the physical limitations of disks, we reluctantly accept them as inevitable. Our goal is to minimize or eliminate the remaining sources of guarantee violations, the avoidable violations. Only these violations can be reduced by improving the scheduling policy. Therefore, we choose to design our system to monitor the behavior of workloads and make the appropriate tradeoffs among the workloads to provide guarantees to workloads whose accesses can be supported within their assigned resource allocations. We avoid diverting resources away from supportable workloads to workloads that are experiencing fundamental violations, because doing so could trigger avoidable violations. But, we also avoid diverting excessive resources away from workloads undergoing fundamental violations. They continue to receive their maximum fraction of resources, to allow them to move past the period of unsupportable behavior quickly.

4 Implementation

This section details our algorithm and its implementation in a user-level scheduler.

4.1 Workloads and requests

As requests arrive at our scheduler, they are tagged with a workload identifier. A workload is the unit to which a guarantee is assigned; the guarantee applies to the set of requests received from that workload. When a new workload enters the system, it requests an unused identifier and specifies its bandwidth requirement in MB/s and its maximum fraction of resources expected. Although we do not discuss it further, nothing in our design or implementation precludes changing the guarantee of an existing workload.

4.2 Round-robin timeslicing

The scheduler executes rounds of timeslices, where each round includes a single timeslice for each active workload. As a system property, a target round length is selected and represents the total time period over which each round executes. The appropriate round length can be calculated using the number of workloads and the "context switching cost" (cost of switching between workloads on a particular disk) to achieve a desired efficiency level, such as 90%. Rounds are repeated indefinitely while the storage system is running. Each round, however, need not be identical to the others.

4.2.1 Beginning a round

At the beginning of a round, the system determines whether the previous round ended early. If so, this represents slack caused by workloads receiving their guarantees in the previous round before using their maximum fractions of resources, and this slack can be exploited in future periods to potentially help workloads that are not meeting their guarantees. Next, a slack aging algorithm is applied; allowing slack to accumulate without limit and then be applied to a single workload can cause starvation for the others. Our algorithm only permits slack from the last two rounds to be carried forward; this number was determined empirically to work best for our test workloads.

Next, we scan the list of workloads and determine which failed to meet their bandwidth requirements in the previous period despite having sufficient requests ready to issue to the disk. These workloads are marked as *unhappy*, and the remaining workloads are marked as *happy*. We then place the happy workloads into one set and the unhappy workloads into another, and randomly permute the ordering of workloads within the sets. This avoids consistently subjecting any one workload to the effects of executing first or last in a period. Then, we concatenate the permuted lists into an ordering of timeslices, where all the unhappy workloads execute first.

The next step is to choose appropriate timeslice lengths for each workload in the round. These initial allocations are predicted to allow each workload to meet its guarantee, but as described later, the algorithm may refine these allocations after timeslices are underway based on real-time observations of workload performance. For each workload, we calculate the amount of data it must transfer in one round to fulfill its guarantee. For instance, if the round length is 2 s, and a workload is guaranteed 1 MB/s, then it must transfer 2 MB in a round. We then reserve for each workload the projected amount of time needed to transfer the appropriate amount of data, based on estimated per-request time. Next, we sum the reservations across the workloads.

The total reservation for the round may exceed the round length; if so, we must start reducing reservations. We start with the last workload scheduled in the round (typically, the happy workload chosen last) and reduce its reservation until the total of all reservations matches the target round length, or the workload has been reduced to its maximum fraction of the round, whichever allows for a longer timeslice. If the total reservation still exceeds the target round length, we continue the process with the $n-1^{st}$ workload, and so on.

Finally, if there is at least one unhappy workload, we attempt to reserve extra time for it, in the hopes of making it happy this period. This may allow for brief periods of lower bandwidth to be averaged out. We choose the first workload in the timeslice ordering (a randomly chosen unhappy workload), and increase its reservation to use any time in the period that has not yet been reserved. This extra time is available only if the initial reservations for the workloads summed to less than the target round size; if it was necessary to reduce the reservations as described in the previous paragraph, such extra time is not available. Last, if there is slack being carried forward from previous rounds, we also allocate this slack to the first (unhappy) workload.

4.2.2 During a timeslice

When a workload's timeslice begins, all requests from the preceding workload have completed and the disk is ready to be dedicated exclusively to the new workload. During the timeslice, available requests (received directly from the workload, or dispatched from a queue of postponed requests from the workload) are sent to the disk until we determine it is time to *drain* the timeslice. When we drain a timeslice, we stop issuing further requests and let any requests pending at the disk complete. Once the disk has completed all pending requests, the timeslice ends. We drain a timeslice when either of the following becomes true:

1. The workload has issued enough requests to transfer its required amount of data for the round (e.g., the 2 MB given in the earlier example).

2. The time reserved for the workload has been exhausted. However, if preceding workloads in the round ended their timeslices early, this surplus is made available to the current timeslice, increasing its reservation from the initial calculation. If it also ends early, the surplus is carried forward again. If the surplus is not used up by the last timeslice, it becomes slack in the next round.

When issuing requests to the disk, we do not constrain the number that can be outstanding at a time, with one exception. We do not allow more requests to be in flight than we project would be able to complete by the end of the timeslice. However, if these requests complete sooner than expected, the workload is permitted to send further requests.

4.2.3 Prediction

The prediction of how long of a timeslice a workload needs is based on a prediction of the time it takes to complete a request at the disk. In an early version of our implementation, we used an algorithm that learned this value for each workload and provided hysteresis via a moving average. Unfortunately, performance and the number of guarantee violations were relatively poor. The workloads we use for testing and evaluation have high variability, with some requests completing in 0.1 ms and others taking two orders of magnitude longer. Just when our algorithm had adapted to quick requests from one of the workloads, the workload began to issue slow ones, and we allowed it to exceed its timeslice. We changed the parameters of the hysteresis across a wide range, with none of the settings proving satisfactory.

Then, we tried using a constant value that represented an estimate of workload behavior that was in between the extremes. Surprisingly, this resulted in much better performance and guarantee adherence. Instead of trying to adapt to a constantly-changing value and often being two orders of magnitude wrong, our "safe" estimate bounds error closer to one order of magnitude. While this is still significant error, our experiments show that it is manageable by our algorithm. The ability to carry forward slack from timeslice to timeslice and across rounds, and to proactively change control decisions within a timeslice, provides enough ability to compensate for imperfect predictions in practice.

5 Evaluation

To evaluate guarantee adherence for our scheduling algorithm and for other proposed bandwidth guarantee algorithms, we ran a series of experiments with combinations of realistic workloads and appropriate guarantees. We observed when guarantees were and were not being met, and quantified which violations were fundamental violations. These results show that our algorithm avoids nearly all guarantee violations other than fundamentally unavoidable ones, while the other algorithms did not deliver consistent guarantee adherence.

5.1 Experimental setup

This section describes the hardware, software, traces, and combinations of workloads we used in our experiments.

5.1.1 Hardware

Each experiment used a single machine with a Pentium 4 Xeon processor running at 3.0 GHz. A dedicated disk was used to host the data being accessed by the workloads; it was a Seagate Barracuda 7200 RPM SATA drive, 250 GB in size. Two other, identical disks stored the OS and traces. All the drives were connected through a 3ware 9550SX controller, and both the disks and the controller support command queuing. The machines had 2 GB of RAM and ran Linux kernel version 2.6.26.

5.1.2 Software

The described algorithm, a baseline that does not provide guarantees or manage workloads at all, and alternatives from the literature were implemented in a disk time scheduler that manages requests coming from a trace replayer. The trace replayer performs simple as-fast-as-possible trace replay from trace files against the standard POSIX filesystem interface. For instance, a read in the trace will result in a standard POSIX `read()` call for the specified region of the local hard drive. The trace files recorded NFS-level read and write accesses to file systems. Files are specified as filehandles; thus, for each accessed filehandle in the traces we assigned a corresponding, appropriately-sized region on the disk. The replayer maintains a specified level of concurrency while replaying the traces; for these experiments, we used 16 threads per workload.

The scheduler receives requests from individual workloads, queues them, and dispatches them to the disk (by making standard I/O system calls) when appropriate. We chose a user-level implementation for simplicity of experimentation and debugging. It would be straightforward, however, to add this scheduler as one of the selectable block I/O schedulers in the Linux kernel. Similarly, it could be incorporated into the kernel of another OS, or used to manage requests being handled by a user-level storage server.

5.1.3 Sharing policies

We implemented four algorithms in our disk time scheduler to compare in our evaluation. The first, *no throttling*, does not manage the workloads or requests in any way. *Token bucket* implements a classic token-bucket–based throttling system attempting to maintain bandwidth guarantees. The bucket size (which represents the maximum surplus a workload can "save up" and use for later bursts) was 5 s worth of bandwidth at each workload's configured rate. p*Clock* is an implementation of the *p*Clock algorithm [4], which uses both token-based metering and deadlines for latency control; we configure it to use a maximum latency of 2400 ms. *Timeslices* is the algorithm described in Sections 3–4 with a fixed round length of 2400 ms.

5.1.4 Workload traces

To evaluate the performance guarantee techniques with realistic workloads, we used traces collected from the day-to-day operations of a feature-film animation company and released by Anderson [1]. To our knowledge, these are the most intense NFS traces that are publicly available today. The traces contain the storage accesses that occur from different machines in a "render farm" as they perform the 3D rendering necessary to generate an animated film.

Each activity record in the traces specifies the IP address of the machine making the request. Thus, we are able to separate accesses from different machines in the render farm, which are presumed to be performing unrelated jobs at a given point in time. The set of requests issued from a particular source machine is, therefore, treated as a workload for the sake of insulation from other workloads and as the unit to which performance guarantees are assigned. We chose ten of the most active machines from the traces to use as our workloads. We then chose a suitable span of time from the traces that is short enough to be able to run a series of experiments but long enough to capture interesting variability in workload behavior.

Because workloads in a datacenter generally use storage systems built from more than one disk, but our evaluation is done on a single disk for simplicity, we scale down the traces by a factor of four in a manner similar to striping. Since we are evaluating the performance of a shared server, we also suppress from the traces those accesses that would be absorbed by a client buffer cache that is 512 MB in size and implements an LRU replacement policy.

The resulting trace excerpts from each workload take approximately thirty minutes to run on our system. We derived appropriate performance guarantees by running each workload separately and examining the full-system bandwidth achieved for that workload's access patterns over time. We then chose a level of bandwidth that would be generally achievable while sharing the system efficiently and fairly with a few other workloads. Each workload, however, exhibits dramatic variability in access patterns, and for each workload we chose a guarantee level that was reasonable for the majority of its run, but could not be met 100% of the time, even on a dedicated disk. We believe it is important to include such behavior in our experiments, to evaluate how the performance guarantee techniques handle fundamental violations.

All but one workload is assigned a 0.2 MB/s bandwidth guarantee; the other, a 0.1 MB/s guarantee. While these bandwidth levels may seem undemanding, recall that we have filtered out requests that would hit in the client's cache from our traces, and note that the data sheet for our disks implies an average throughput of about 82 IOPS for random requests (based on average seek time and rotational latency). For an example request size of 4 KB, single workloads using a dedicated disk are therefore expected to receive 0.328 MB/s. Hence, our guarantee levels are appropriate and well-proportioned for our disks.

5.1.5 Workload combinations

For our experiments, we ran combinations of workloads consisting of between 2–10 workloads chosen from among the ten trace excerpts we generated. For two-workload sets, there are 45 combinations (not including pairs of the same workload), and we ran experiments with all of them. For nine-workload sets, there are ten combinations; and there is only one ten-workload set. We ran experiments with all of these combinations as well. To keep total experiment time manageable, however, we sampled the > 45 combinations of workloads sized between 3–8, and randomly chose a subset of 45 combinations for each of these sizes. The same random selections were used for each of the sharing policies we evaluated.

5.1.6 Measurements

We ran each of the ten workloads alone to measure the performance they receive when using a dedicated disk. This allows us to identify the parts of the traces where fundamental violations occur. Recall that fundamental violations are those violations that could not have been avoided by an efficient and fair system. For a window of time, if a workload is able to receive x MB/s on a dedicated disk, and when it shares a disk it is assigned a maximum fraction f, then a fair system operating at efficiency level R (using our nomenclature from Argon [16]) should be able to provide bandwidth $\geq f \times R \times x$ in the same window. We measure bandwidth during 15-second windows.

Next, we run each of the different combinations of workloads. If, when a workload is sharing the disk, we detect a bandwidth violation, we can identify the corresponding window from the dedicated-disk run of that workload. By plugging in the appropriate values of x and f and expecting a system efficiency of $R = 0.9$ (as we generally achieve with Argon), we can discern whether the violation in the shared run is fundamental or avoidable.

5.2 Results

This section presents the results of our experiments. We first show the bandwidth achieved when a single combination of workloads runs to illustrate the difference between fundamental and avoidable violations. Then, we examine the violations experienced over the entire set of workload combinations.

5.2.1 Example timelines

We plot the bandwidth received by one workload over time as it shares the disk in one of the randomly-chosen combinations of five workloads. The most straightforward way to depict bandwidth over time is to place successive windows of time on the x-axis and bandwidth on the y-axis. Unfortunately, in this format, it is difficult to locate corresponding periods in a trace across different runs, because the x-axis represents time rather than location in the trace — and thus is affected by performance. Instead, we graph successive portions of the trace on the x-axis, and the amount of time it took to replay that part of the trace on the y-axis. If a workload has been assigned a guarantee of 1 MB/s, then we plot successive 15 MB windows of the trace (corresponding to an expected 15 second window of time) on the x-axis, and the average number of seconds it took to transfer 1 MB during each of the windows as the y-values. In this representation, a violation occurred whenever the plot is above the line $y = 1$ second, rather than below the line $y = 1$ MB/s, as would be the case in the conventional way of graphing the data. The advantage to this format is that points on the x-axis line up from run to run, regardless of performance.

Figure 1(a) shows the performance the workload receives with a dedicated disk, but the y-values of the data have been scaled to represent the performance a fair and efficient system running at 90% efficiency and allocating a maximum fraction of 1/5 should be able to provide that workload, as described in Section 5.1.6. This is accomplished by taking the number of seconds it took to transfer the data with a dedicated disk and multiplying by 5 (the reciprocal of the fraction) and then by 1/0.9 (the reciprocal of the efficiency level). By inspecting Figure 1(a), we see that a fair and efficient system should be able to meet the guarantee (points below the line $y = 1$) most of the time, but there are several periods where fundamental violations will occur.

Figure 1(b) shows the performance the workload receives when sharing the disk under the no-throttling policy, where no attempt is made to manage the interference between the workloads. Note that many of its violations occur during the periods where fundamental violations are expected. Some, however, such as the ones at approximately $x = 750$, are avoidable violations because there are no corresponding violations in Figure 1(a).

Figure 1(c) shows the performance the same workload receives under the timeslicing-based scheme described in Sections 3–4. Note that each violation is a fundamental violation in this particular run. Thus, in this specific experiment, our system has succeeded in maintaining efficiency for the workload and eliminated all the artificial or avoidable sources of guarantee violations. In fact, many of the fundamental violations predicted by Figure 1(a) do not occur with the timeslicing scheduler. This is because the fundamental model predicts which violations occur when the fraction of time available to the depicted workload is limited to 1/5. But, as described in Section 4, if another workload meets its guarantee for a round in less than its maximum time, this slack is made available to other workloads. Hence, the windows where we avoid predicted violations are windows where statistical multiplexing works to the favor of the workload shown.

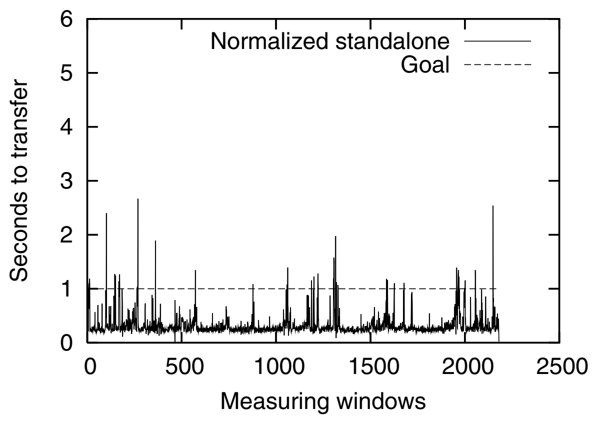
While only one of the five workloads is plotted, the other four experience similar behavior.

5.2.2 Fundamental violations

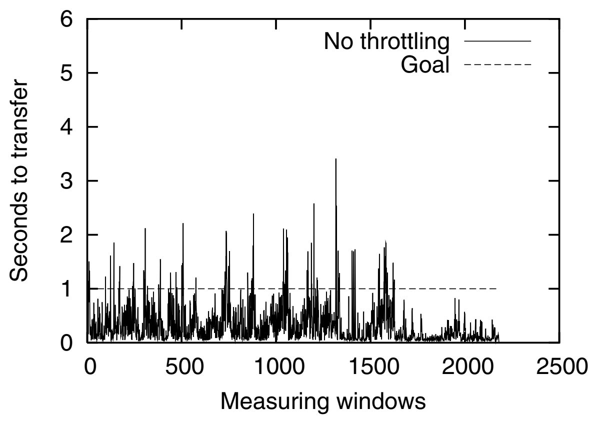
Using the method described in Section 5.1.6, we calculate the expected number of fundamental violations for each workload when assigned 1/2–1/10 of the system. These values correspond to the maximum fraction given to the workloads in our combinations of 2–10 workloads. (Note that neither our design nor implementation preclude other fractions or non-matching fractions among the workloads.) The results are shown in Figure 2. Sets of two or three workloads should fit easily on the disk. Larger sets will have some unavoidable violations, in up to 20% of the periods.

Figure 3 shows conservative bounds on statistical multiplexing derived from Hoeffding's inequality [5]. This equation indicates an upper bound on how many violations should result from the failure of statistical multiplexing — in other words, how often the combination of workloads would, in total, require more disk time than 100%. The bound is based on the mean resource requirements and the range of resources needed by the workloads (e.g., average of 5% but as much as 20% in some periods). The two lines represent the violations predicted using the average range among our workloads and the worst-case among our workloads.

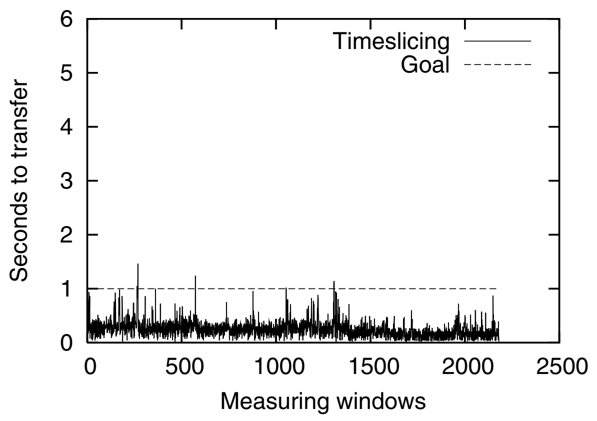
The inequality predicts that there may be periods of over-subscription when the number of workloads exceeds five. This is of significance because when there is such a period, no slack will be available in our system. This will prevent our algorithm from carrying forward extra time between rounds. During periods where our algorithm loses the ability to exploit slack, it loses much of its ability to correct for mispredictions in estimated request times and timeslice lengths.



(a) Normalized standalone



(b) No throttling



(c) Timeslicing

Figure 1: Bandwidth guarantee adherence over time for a dedicated disk and two of the schedulers

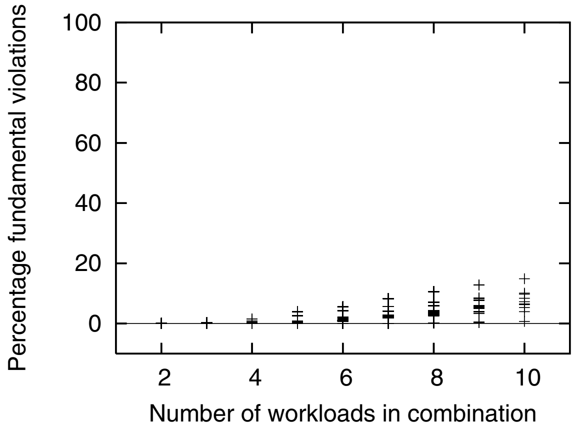


Figure 2: Percentage of fundamental violations

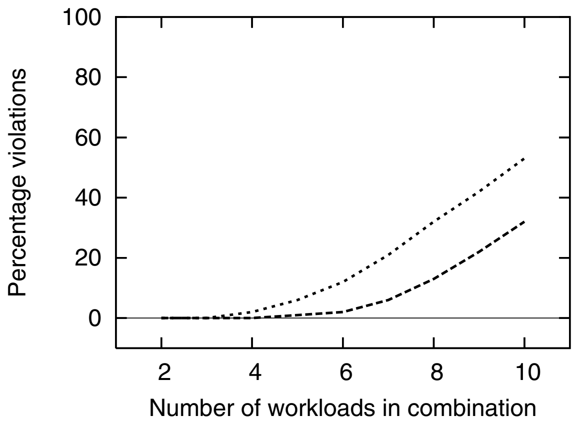


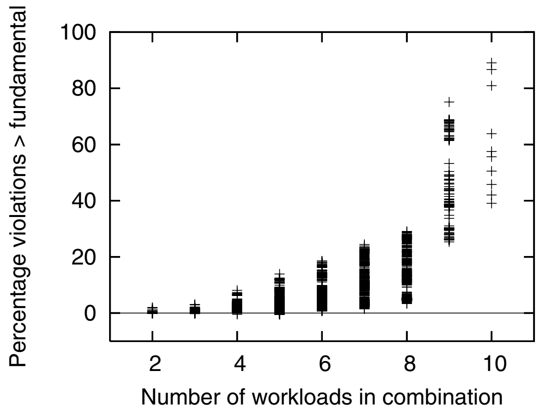
Figure 3: Percentage of violations predicted by Hoeffding's inequality

5.2.3 Combinations of workloads

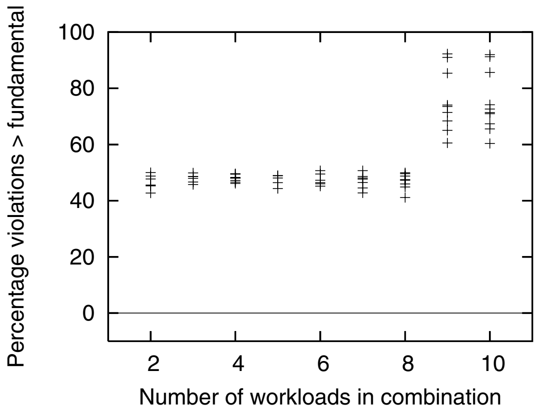
Figures 4(a)–4(d) show violations for each of the schedulers we implemented, running with the sets of workload combinations we generated. In these scatterplots, each point corresponds to the percentage of periods with violations experienced by one of the workloads when running in one of the combinations. The number of periods with fundamental violations, calculated by the method described in Section 5.1.6, is subtracted from the number of observed violations to yield the depicted values.

Without throttling (Figure 4(a)), avoidable violations occur even for the two-workload combinations. When ten workloads share the system, some experience nearly constant violations. Without management of the requests coming from each workload, fairness or proportional sharing cannot be ensured. In addition, the scheduler makes no effort to maintain the efficiency of the system. The benefits of locality that some workloads enjoy when running alone may be lost when their requests are interleved with those from other workloads at a fine-grained level.

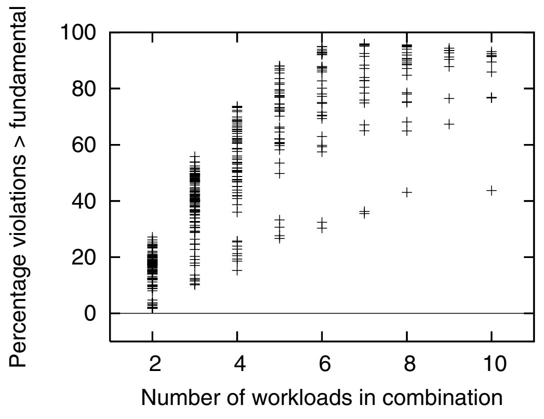
With token-bucket throttling (Figure 4(b)), the situation counter-intuitively becomes worse instead of better. Even for two workloads, avoidable violations occur in nearly half of all periods. While token-bucket schedulers attempt to manage fairness among the workloads, they do so by strictly limiting the number of requests each can send to the system. As described in Section 2.2, token bucket-schedulers work by limiting



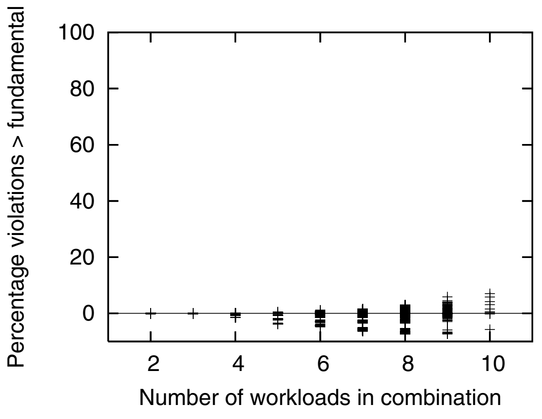
(a) No throttling



(b) Token bucket



(c) *p*Clock



(d) Timeslicing

Figure 4: Percentage of violations above or below the fundamental violations for each of the scheduling policies

each workload to an upper bound: its bandwidth guarantee. This is fundamentally backwards from the intended goal of maintaining a lower bound. Consequently, even if workloads can fit together on a system, they are at best limited to exactly their guarantee level of bandwidth. Thus, the best-case scenario finds workloads constantly on the edge between making their guarantee and suffering a violation. But, token-bucket schedulers finely interleave requests coming from different workloads, disrupting any locality they may have; no attempt is made to preserve their efficiency. This may result in them requiring substantially more time to complete their requests, and there may not be enough time available in the system to service all the workloads at their reduced efficiency. Note that only a limited number of combinations of workloads were tested for this graph, because the run time of experiments with this scheduler was prohibitive.

*p*Clock also counter-intuitively makes matters worse (Figure 4(c)). While we have received acceptable performance when using our implementation of *p*Clock with other workloads, and while it outperforms token bucket for some small combinations, it is not generally effective on these workloads. Like token-bucket throttling, it does not fully preserve the efficiency of workloads when they share a system.

The timeslicing-based technique we proposed in Sections 3–4 eliminates most non-fundamental violations (Figure 4(d)). It does this by explicitly managing both the proportion of the system a workload receives, and the efficiency with which it operates during its assigned fraction of time. This allows workloads that benefit from locality to continue operating at close to their full efficiency, avoiding the increase in required resources suffered by the other systems. In some instances, the number of violations is actually less than the expected number; there are no avoidable violations and fewer fundamental violations than predicted. This is the result of statistical multiplexing working to the benefit of the workloads; sometimes, another workload will meet its guarantee in less than its assigned fraction, and this slack will allow another workload to use resources beyond its assigned fraction. Since the number of fundamental violations is calculated for a workload's assigned fraction, some of those violations may not occur when surplus resources are available.

Unfortunately, a low level of avoidable violations does remain in some cases. This begins at the point where Hoeffding's inequality predicts that the system will be oversubscribed in some periods (Figure 3). When this occurs, no free resources are available in the system, and the scheduler does not have slack available to make up for imperfect control decisions. The scheduler sizes the timeslices for each workload, and determines when to start draining requests and end a timeslice, based on projections of how long a workload's requests will take. The workloads we used in these experiments are highly variable, with some requests completing in tens of milliseconds and some in a tenth of a millisecond. Thus our scheduler, which does not have complex prediction algorithms, cannot make consistently accurate predictions.

Fortunately, slack allows the scheduler to make up for sub-optimal decisions. But, if slack is not available, the effects of imperfect predictions start to show up as "avoidable" violations. We called any violations that would not occur under a fair and efficient scheduler *avoidable*. While the results show that our scheduler is significantly more effective than the others, it is not perfectly fair and efficient because of the uncertainty it faces when trying to control a highly variable workload. These violations might belong in a third class, *prediction* violations. It is not clear to what extent more sophisticated prediction algorithms or a more resilient scheduler would be able to reduce this type of violation.

6 Conclusion

Bandwidth guarantee violations can be dramatically reduced by using a scheduler that explicitly manages interference between workloads. By incorporating the notion of efficiency, violations can be differentiated between those that are fundamental and those that are avoidable. Our timeslicing-based scheduler strictly limits interference while adapting timeslice lengths to control bandwidth. In experiments on realistic workloads, it eliminates almost all avoidable violations, and is able to use slack to reduce the number of "fundamental" violations as well. This allows workloads sharing a storage system to receive predictable, controllable performance.

Acknowledgements

We thank Eric Anderson and the animation company for making the traces we used publicly available. We thank Julio López for help with performance debugging, and Michael Stroucken and Mitch Franzos for help with equipment.

References

[1] Eric Anderson. Capture, Conversion, and Analysis of an Intense NFS Workload. *FAST '09: Proceedings of the 7th USENIX Conference on File and Storage Technologies*, 2009.

[2] John Bruno, Jose Brustoloni, Eran Gabber, Mary Mcshea, Banu Ozden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. *Proceedings of the 1999 International Conference on Multimedia Computing and Systems*, 1999.

[3] David D. Chambliss, Guillermo A. Alvarez, Prashant Pandey, Divyesh Jadav, Jian Xu, Ram Menon, and Tzongyu P. Lee. Performance virtualization for large-scale storage systems. *SRDS '03: Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, 2003.

[4] Ajay Gulati, Arif Merchant, and Peter J. Varman. pClock: An Arrival Curve Based Approach For QoS Guarantees In Shared Storage Systems. *SIGMETRICS '07: Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 2007.

[5] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, **58**(301):13–30, 1963.

[6] Lan Huang, Gang Peng, and Tzi-cker Chiueh. Multi-Dimensional Storage Virtualization. *SIGMETRICS/Performance '04: Proceedings of the 2004 International Conference on Measurement and Modeling of Computer Systems*, 2004.

[7] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. *SOSP '01: Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[8] Tim Kaldewey, Theodore M. Wong, Richard Golding, Anna Povzner, Scott Brandt, and Carlos Maltzahn. Virtualizing Disk Performance. *RTAS '08: Proceedings of the IEEE Real Time and Embedded Technology and Applications Symposium*, 2008.

[9] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance Isolation and Differentiation for Storage Systems. *IWQOS '04: Proceedings of the 12th IEEE International Workshop on Quality of Service*, 2004.

[10] John D. C. Little. A Proof for the Queueing Formula: L = Lambda times W. *Operations Research*, **9**(3):383–387, 1961.

[11] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Facade: Virtual Storage Devices with Performance Guarantees. *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[12] Anastasio Molano, Kanaka Juvva, and Ragunathan Rajkumar. Real-Time Filesystems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach. *Proceedings of the 18th IEEE Real-Time Systems Symposium*, 1997.

[13] Anna Povzner, Tim Kaldewey, Scott Brandt, Richard Golding, Theodore M. Wong, and Carlos Maltzahn. Efficient guaranteed disk request scheduling with Fahrrad. *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008.

[14] Prashant Shenoy and Harrick M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. *SIGMETRICS '98: Proceedings of the 1998 International Conference on Measurement and Modeling of Computer Systems*, 1998.

[15] Jonathan S. Turner. New Directions in Communications (Or Which Way to the Information Age?). *IEEE Communications Magazine*, **24**(10):8–15, 1986.

[16] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: performance insulation for shared storage servers. *FAST '07: Proceedings of the 5th USENIX Conference on File and Storage Technologies*, 2007.

[17] Theodore M. Wong, Richard A. Golding, Caixue Lin, and Ralph A. Becker-Szendy. Zygaria: storage performance as a managed resource. *RTAS '06: Proceedings of the IEEE Real Time and Embedded Technology and Applications Symposium*, 2006.

[18] Jianyong Zhang, Alma Riska, Anand Sivasubramaniam, Qian Wang, and Erik Reidel. Storage Performance Virtualization via Throughput and Latency Control. *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2005.