Virtualizing Disk Performance

Tim Kaldewey, Theodore M. Wong†, Richard Golding†, Anna Povzner, Scott Brandt, Carlos Maltzahn
Computer Science Department, University of California, Santa Cruz
†IBM Almaden Research Center
{kalt, tmwong, golding, apovzner, scott, carlosm}@cs.ucsc.edu

Abstract

Large- and small-scale storage systems frequently serve a mixture of workloads, an increasing number of which require some form of performance guarantee. Providing guaranteed disk performance—the equivalent of a "virtual disk"—is challenging because disk requests are non-preemptible and their execution times are stateful, partially non-deterministic, and can vary by orders of magnitude. Guaranteeing throughput, the standard measure of disk performance, requires worst-case I/O time assumptions orders of magnitude greater than average I/O times, with correspondingly low performance and poor control of the resource allocation. We show that disk time utilization—analogous to CPU utilization in CPU scheduling and the only *fully provisionable aspect of disk performance—yields greater control, more efficient use of disk resources, and better isolation between request streams than bandwidth or I/O rate when used as the basis for disk reservation and scheduling.*

1 Introduction

Significant research has been conducted on managing and guaranteeing CPU performance. Although many applications—e.g., multimedia, transaction processing, real-time data capture, and virtual machines—require I/O performance guarantees, relatively little successful research has been conducted on guaranteeing storage performance for mixed workloads.

As traditional real-time applications such as multimedia become ubiquitous, managing storage performance is critical for storage systems concurrently executing a mix of workloads. In small dedicated devices (e.g., an iPod), performance can be managed via offline or design-time resource provisioning. For other devices, including desktop machines and, especially, large shared storage servers, some other mechanism is needed.

Our goal is to provide a "virtual disk" in a shared storage system: applications should be able to reserve disk performance and the system should guarantee that performance, as shown in Figure 1. Doing so requires a uniform representation of disk performance that can be guaranteed regardless of what other applications are accessing the disk or how they behave.

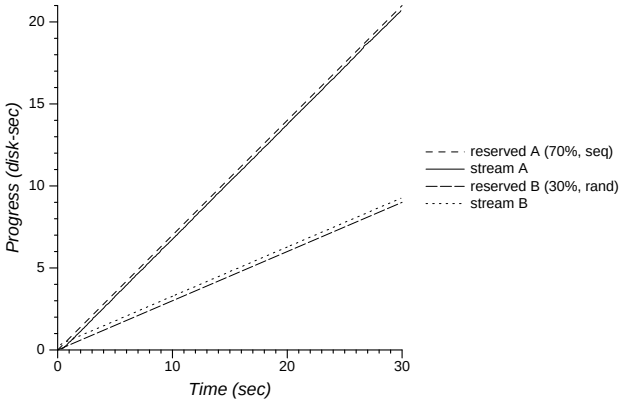


Figure 1: Progress of two I/O streams; one random and one sequential. Each stream uses its own virtual disk, both of which are hosted by a single physical disk. The virtual disk of the sequential stream has reserved 70% of the physical disk time and the virtual disk of the random one has reserved 30%. Other combinations (not shown) are similar.

Disk throughput has traditionally been used to describe storage system performance. Although throughput is a natural way to represent application requirements, guaranteeing throughput is difficult for four reasons. First, disk requests are stateful: the time required for a request depends heavily upon the location of the immediately preceding request. Second, disk request times are partially non-deterministic: disks are intelligent devices that perform bad block remapping, re-calibration, and other operations that can affect I/O times in ways that are impossible to determine *a priori*. Third, disk requests are non-preemptible: once issued to the disk, a request must be allowed to finish. Fourth, and perhaps most significant, the impact of the previous three is made critical by the huge difference between best- and worst-case request times: a best-case request can take a fraction of a millisecond, while a worst-case request may take tens of milliseconds. Thus throughput depends

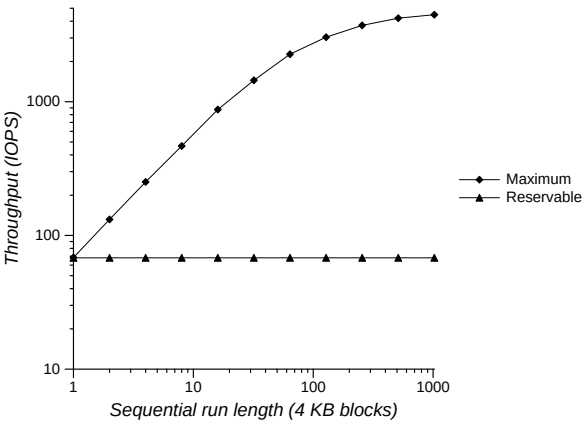


Figure 2: Reservable and maximum I/O rate for workloads, with increasing amounts of sequentiality. The difference between the two curves represents the opportunity for improved resource reservation.

heavily upon application I/O behavior, which is beyond the control of the storage system.

One potential solution to this problem is to reserve performance based on worst-case request times. However, because of the vast difference between best- and worst-case times, less than 1% of achievable disk performance is reservable. Figure 2 shows the opportunity cost of managing disk performance using throughput, with reservable performance orders of magnitude lower than actual achievable performance.

Providing a virtual disk and guaranteeing performance in a shared storage system requires isolating each application from the behavior of others. Head seeks from one region of a disk to another are the primary source of delay in serving requests and each request stream must be correctly charged for its seek behavior in order to avoid interfering with the performance of other streams. Ideally, each application should behave as though it were the only one running on a device with fractional performance equal to its share of the actual disk.

We show that *disk time utilization*—time spent servicing I/O requests, analogous to CPU utilization—is the appropriate basis for managing disk performance. Unlike throughput, disk utilization is a quantity that we can actually guarantee without losing most of our performance. Because we can correctly account for disk time, we can charge each stream for the disk seeks it incurs, allowing isolation between application request streams. Finally, a virtual disk comprising a time share of the actual disk can be requested, feasibility can be verified ($\sum$ reservations $\leq$ 100%?), and the share can be guaranteed without any assumptions about application behavior. As shown in Figure 1 and discussed further in §4, managing disk performance with utilization allows a clean allocation of disk performance independent of application behavior, exactly as desired for a virtual disk.

In fact, disk time utilization is the *only* workload-independent way of expressing and managing disk performance[1]. The only thing that a disk guarantees is that it will spend time servicing I/O requests and thus the only aspect of I/O performance that can be provisioned is time on the disk. All other disk performance metrics are dependent upon workload, data layout, and disk parameters.

For the rest of the paper we will refer to disk time utilization simply as "utilization". We show how to provide virtual disks with a utilization-based disk scheduler and compare it to a throughput-based approach.

2 Virtual disks based on utilization

We define a virtual disk in a shared storage system as a fractional share of a disk whose performance depends only upon its workload, its share, and the base performance of the device. Importantly, the performance of a virtual disk should be independent of the performance or behavior of any other. We argue that utilization is the appropriate basis for providing a virtual disk. Disk time is a quantity that we can guarantee while keeping most of the efficiency of the raw device. Within a given disk time utilization, the performance of a request stream will be determined solely by its behavior. In contrast, by guaranteeing I/O rate one cannot exploit the increased efficiency of well-behaved (*i.e.*, sequential) streams as the guaranteeble performance is constant and must be based on worst-case assumptions.

When a disk is shared by multiple workloads, one workload can potentially degrade performance of other workloads. Isolation between request streams is critical; an application running on a virtual disk should receive the same performance regardless of the behavior of other applications accessing the same shared disk. Managing disk performance in terms of utilization allows us to correctly account for the seeks incurred by each I/O stream, allowing isolation between request streams. Finally, managing virtual disks based on utilization is easy because utilization is additive and feasibility can be trivially verified, so the share can be guaranteed without any *a priori* information about application behavior.

The rest of this section discusses these points in more detail. We also show how utilization can be used to guarantee application performance requirements.

2.1 Measuring and guaranteeing utilization

We define utilization using the standard queuing-theoretic model. We assume that a disk services one request at a time and model it as a single queue/single server system. Utilization is thus the fraction of time that the disk spends executing I/O requests, in units of μseconds of execution

[1] "Always bear in mind that the real measure of computer performance is time." Computer Architecture: A Quantitative Approach, p. 37, by Hennessy and Patterson, Morgan Kaufmann Publishers, Inc., 1990.

per second. Each I/O request can be thought of as consuming a certain number of μseconds of the available execution time. This definition is correct for single-server systems; it requires extension to handle multiple-server queuing systems, such as RAID arrays with multiple disks operating in parallel.

While utilization is a measure of usage over a given period of time, the I/O scheduling algorithm needs estimates of the "instantaneous" utilization received by different I/O streams to determine which I/O requests to schedule and which to delay. There are several ways to compute the estimates, all of which use execution time over some short period of recent history, possibly with age-weighting. These estimators necessarily exhibit some lag, which must be considered in the decisions that use the estimates.

At the lowest level, the I/O scheduler decides which I/O request to send to the disk for service in order to enforce utilization reservations. The scheduler also has a secondary goal of maximizing disk efficiency by minimizing disk head movement. The scheduling decision is generally based on three things: what requests are enqueued and can be chosen for service; how each virtual device is doing with respect to its reservation; and the estimated utilization needed for each pending request. The cost of executing a request depends on the state of the disk, including the rotational position of the platter and the seek position of the head.

Figure 3 shows the overall flow of the I/O scheduler. There are three major events in the lifecycle of an I/O request: scheduling; execution; and completion. The decision to schedule one request or another depends on the *accounting* for different virtual devices—for example, whether a virtual device has reached a performance limit and thus its requests are temporarily blocked, or if it is below its reservation and needs to execute additional requests.

This model raises three questions. First, the scheduler requires an estimate of the execution time of each I/O request. How do we determine these? Second, the scheduler needs to know the actual execution time of each I/O request. How are these measured? Third, I/O workloads can change abruptly. How do we avoid unstable behavior when this occurs?

Before executing a request, the scheduler can only predict how long its execution will take. In our system, we estimate the execution time e when a request is scheduled. There are several possible ways to compute the estimate, ranging from using a constant "safe" value, such as the average execution time of a random I/O request, to modelling the detailed behavior of the disk, as has been done for SPTF head schedulers [15]. Detailed modelling can be difficult, since every disk is different, and so we take a simple approach where sequential I/O requests are estimated at a small constant value and other requests are estimated at a safe value.

After the completion of each request, the scheduler corrects its accounting based on the measured (actual) execution time. When the request completes, the scheduler computes the actual execution time a as the time between the completion of the previous request and that of the current request $c_i - c_{i-1}$, or, after an idle period, between the release and completion of the current request $c_i - r_i$.

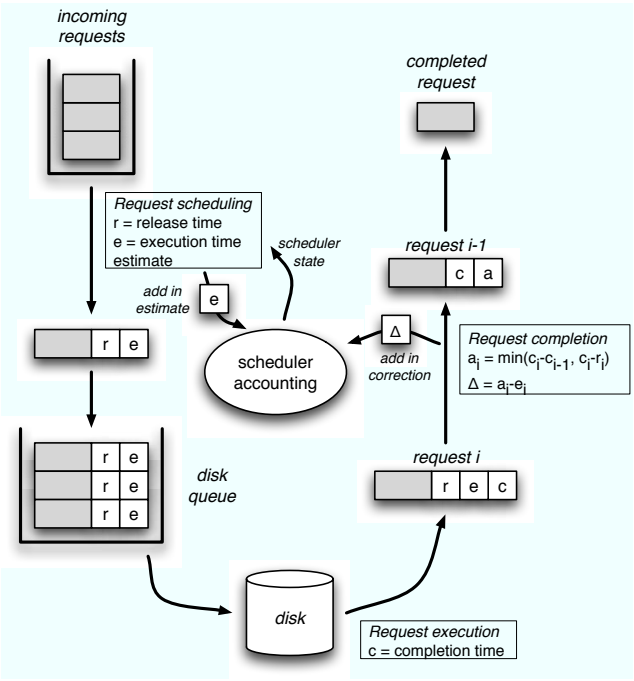


Figure 3: Flow of I/O requests through an I/O scheduler.

The scheduler uses the measured execution times to compute an estimate of the utilization recently obtained by each virtual device. We use *token buckets* to track how far each virtual device has progressed with its reserved utilization (see §3). The token bucket fills at a constant rate—the rate at which the virtual device is supposed to utilize the disk—and the scheduler works to keep the bucket level in a range $0 \pm u$, where u is the expected utilization over a moderate period p (for example, one second). For any period at least $2p$ long, the scheduler will work to ensure that each virtual device gets its proper utilization, but over short durations any virtual device can get up to u ahead or behind the long-term average.

Finally, as illustrated by the time series in Figure 11 (§4.4), a scheduler must cope with abrupt changes in the offered load. Since the scheduler is, in effect, a control system that is attempting to ensure that each virtual device's reservation is met, the scheduler must be designed to control oscillatory behavior and the effects of transient load changes.

2.2 Providing isolation

Isolation between virtual devices is critical. Our system provides isolation in five ways. First, and most importantly, we guarantee to each virtual device the utilization it has reserved. Second, we account for all seeks and other

delays. Third, we charge each virtual device for the time spent on each of its requests. Fourth, we schedule I/O requests with sufficiently coarse granularity that inter-stream seeks are minimized and the time spent on each stream's requests is largely due to its own intra-stream seek behavior. Finally, any cost of the remaining inter-stream seeks is randomly distributed across the active virtual disks, representing a small relatively constant overhead on each virtual disk's performance which can be mitigated by leaving a small fraction of disk utilization unreserved.

2.3 Admission control

The scheduling flow in Figure 3 assumes a reservation for each virtual disk. The system must be able to properly admit or deny requests for new virtual disks or changes in virtual disk parameters. Utilization allows for a simple admission control decision: each disk has 100% allocatable utilization, and any set of reservations that sum to no more than 100% are feasible. We note that utilization has been used for decisions similar to admission control in other projects, such as the short-term utilization measure in the Minerva system [1, 2].

2.4 Translating application requirements

Although not strictly required for a virtual disk, the isolation they provide allows soft performance guarantees in terms of throughput. Each application has its own requirements and behavior which can be translated into a utilization required to provide the desired performance given that behavior. Although we do not focus on the translation in this paper—it has been explored in other work, such as by Borowsky *et al.* [2]—we discuss it briefly here.

The general approach is determined by the physics of disk drive operation. When a disk services a sequence of I/O requests from one application, it must first seek the head to the proper track, possibly wait for the proper sectors to pass under the head, and then transfer the data in the sectors to memory buffers for transmission to the host. This leads to a model for translating application requirements and behavior into an expected utilization. We give a simple example; others have developed more complete models [2].

Consider a multimedia application that issues runs of strictly sequential requests: its performance depends on how many blocks the disk services sequentially before moving the head elsewhere. That is, the time to process a sequence of n consecutive block requests is approximately $t = 0.5s_{\max} + 0.5p + nx$, where s_{max} is the worst case seek time, p is the rotation time and x is the time to transfer one block off of the media. (The effective average rotational delay may be lower on some disks when most of the blocks are being read on one track; this formula ignores track-switching costs.) The number n depends on how many requests the application makes available for the scheduler to process together, and how many of those requests the scheduler actually executes sequentially before scheduling requests from other virtual devices. The application can know the number b_a of sequential requests it will make, but the degree of interference between sessions depends on the scheduler. In practice, the scheduler can enqueue a maximum of b_s sequential requests, dependent on granularity (one second, in our scheduler). Thus for an I/O rate r, the application should reserve utilization $u = r/(0.5s_{\max} + 0.5p + \min(b_s, b_a)x)$.

This example focuses on I/O rate as the application's expectation. Other systems, such as Façade [14] and SLEDS [6], focused on latency and, in effect, maintain latency goals using a dynamic control system to adjust the utilization given to each session.

3 Implementation

We implemented a virtual device driver using utilization based I/O scheduling to investigate the virtual disk approach described in §2. The driver was derived from our previous Zygaria driver [27], which provides medium-term average performance guarantees using I/O rate as a common metric, which we replaced with utilization.

Zygaria is implemented as a Linux kernel module layered on top of an existing block device. It exports a set of virtual block devices ("virtual disks"), one for each I/O request stream. The driver provides system calls for setting lower and upper bounds (*reserve* and *limit*) on the performance of the virtual block devices, as well as for querying their parameters and performance measurements. The system calls also perform admission control to ensure the requested reservation is feasible.

The major difference between the old and new Zygaria drivers is how they measure resource usage: The old driver uses the number of I/O requests, while the new one uses execution time measured in μseconds. Both use similar data structures to track resource usage.

For each virtual device, the driver maintains two "token buckets", one for the reserve and another for the limit. Both fill at a rate equal to their fractional share of the overall disk time. The reserve bucket represents the guaranteed (reserved) share of disk time, while the limit caps the use of unclaimed resources (slack). The first request in a virtual device's queue is assigned a deadline that is equal to the time at which the virtual device's reserve bucket will have enough "time" to satisfy the request.

Based on a weighted moving average of the resource usage of each virtual device over the last few seconds the scheduler determines which device should get scheduled when there is slack in the schedule. The limit bucket regulates by how much a virtual device can exceed its reservation.

The scheduling algorithm determines which I/O request to schedule next as follows. First, it eliminates any virtual

devices that are at or above their limit or that have no I/O requests enqueued. Among the remaining virtual devices, it does EDF scheduling: it determines which virtual device has the earliest deadline. If there is slack in the schedule, the scheduler selects the virtual device with the lowest usage.

When a virtual device is selected, its I/O request is sent to the low-level device driver and the accounting for the virtual device is updated to reflect the request. The driver maintains a number of requests outstanding at the low-level driver (up to 20 requests for the throughput-based version). This allows the low-level driver to reorder requests in order to improve head scheduling efficiency, while bounding the lag between the decision to schedule a request for execution and the actual execution.

Because this algorithm updates a virtual device's accounting data structures when an I/O request is scheduled, the scheduler can only use an estimate of the request's execution time. The actual execution time is not known until the request has completed. This is a non-issue when scheduling using I/O rate, since each request uses exactly one resource unit.

As shown in Figure 3, Zygaria initially decrements the bucket level using an estimate, e, and then corrects the accounting when the request completes execution. The driver keeps track of the initial estimate e and the time that the request was sent for execution, r. When the request completes, the actual execution time a is computed as either the time from the completion of the previous request to the completion of this request, or the time from scheduling the request to its completion if the disk was idle when the request was issued. The driver computes the difference $\Delta = a - e$ and updates the accounting accordingly, issuing more requests if the remaining resources are sufficient to allow it.

The simplest way to estimate how long an I/O request will take is to assume a single worst- or average-case execution time for every request. This is unsatisfactory when an application is performing sequential I/Os: on one disk, we found that sequential requests typically execute in less than 10 μsec, while random requests typically take around 15 msec. This implies that the scheduler should charge the same amount of estimated execution for more than 1500 sequential operations as for one random operation. The algorithm in Zygaria would not enqueue enough sequential I/O requests to get good head scheduling efficiency if it treated sequential requests the same as random ones.

Several approaches exist to estimate the execution time of I/O requests. The Freeblock disk scheduler [15] includes a detailed model of the disk drive in the device driver, and uses that to generate detailed, accurate predictions. Various machine learning techniques have been evaluated to estimate storage workload behavior [24].

To minimize overhead we have chosen a simpler approach. Each request is categorized as random or sequential, based on the distance between the request's block address and the previous one. We use a fixed estimate for each category, 20 msec for "random" requests ($\geq$ 100 sectors from the previous request) and 300 μsec for "sequential" requests ($<$ 100 sectors from the previous request). We evaluate this approach in §4.3.

4 Evaluation

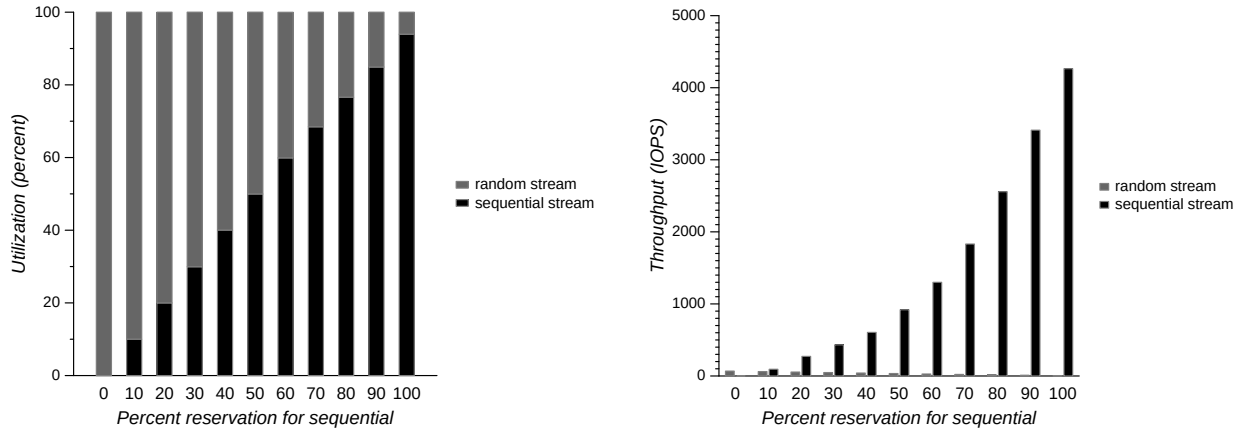
We compared the utilization-based version of Zygaria with our earlier throughput-based version to evaluate their ability to provide the performance guarantees and isolation required for a virtual disk. A virtual disk must guarantee the availability of the resource, independent of application behavior. As with a physical disk, dependencies among multiple applications affecting their storage access behavior are beyond the control of the storage system. The first set of experiments (§4.2) validates the argument that utilization allows better control of performance than does throughput. The second set of experiments (§4.3) compares the sensitivity of each scheduler to the accuracy of its disk drive model, showing that the utilization-based version works correctly with rough models, while the throughput-based scheduler is sensitive to errors in the model. Finally, the third set of experiments evaluates the time-varying properties of the utilization-based scheduler (§4.4). The overhead of the Zygaria scheduler has been evaluated previously [27] and found to be negligible, hence scalability is not an issue. Our results demonstrate the suitability of a utilization-based scheduler as the basis for a virtual disk.

4.1 Experimental setup

Our experiments were conducted on an IBM x335 server with a 2.4 GHz Xeon processor and 512 MB of main memory, running Fedora Core 5 Linux (kernel version 2.6.17). We used an IBM Deskstar 34GXP ATA disk drive (20.5 GB, 7200 RPM, 3.5 in) for all tests; the disk has a measured nominal (worst-case) throughput of 60 IOPS. To avoid interference from other tasks (*e.g.*, operating system maintenance and other applications), the disk was used as an unmounted raw device dedicated to the experiments. We used the modified version of the Zygaria disk scheduling driver described in §3 (and the original one [27] for comparison).

We used the Pharos workload generator [27] to generate block I/O requests. Pharos produces random or sequential I/O streams of 4 KB requests with a variety of different temporal arrival patterns; we used both open-loop arrivals with a constant interarrival time and closed-loop arrivals with constant think time and a fixed number of outstanding requests.

Utilization is reserved by virtual devices which serve streams of I/O requests. Request streams themselves receive utilization and performance while running on virtual devices. For brevity, in the following discussion we talk



(a) Utilization

(b) Throughput

Figure 4: Controlling performance using utilization: utilization and throughput of one random and one sequential request stream from the utilization-based scheduler as the reservation of the sequential stream varies from 0% to 100% of available utilization (and the reservation of the random stream varies from 100% to 0%).

about request streams reserving and achieving utilization. It should be understood that wherever we refer to a request stream's reservation, we are actually referring to a request stream whose requests are served by a virtual device with that reservation.

4.2 Controlling performance

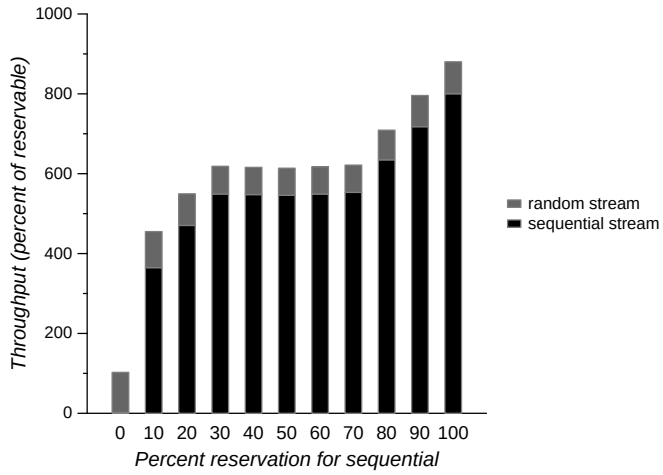
We first compared the utilization-based scheduler with the earlier throughput-based scheduler to determine which provided better control over resources. By better control, we mean the ability to make reservations that are both fulfillable and that effectively determine the division of performance. A fulfillable reservation is one that the scheduler can actually deliver across a wide range of conditions (such as other interfering workloads). When the system is under heavy load, the reservations should determine the performance of the virtual devices and not, for example, fair sharing policies.

We conducted a simple set of experiments to evaluate these aspects of the two schedulers. The experiments used two virtual devices—one serving sequential requests, the other, random requests—together reserving all reservable resources of the disk. We varied the amount of resources reserved for the two virtual devices and measured the resulting performance (utilization or throughput) of the request streams they served, as shown in Figures 4 and 5. For each virtual device, a workload generator sent I/O requests as fast as possible to keep the request queues non-empty. For the sequential stream, the generator sent 1024 sequential requests before sending a request at a new random location. Each experimental run took 85 seconds to determine steady-state performance for a given set of resource reservations. While we repeated each experiment many times, the differences between runs were indistinguishable.

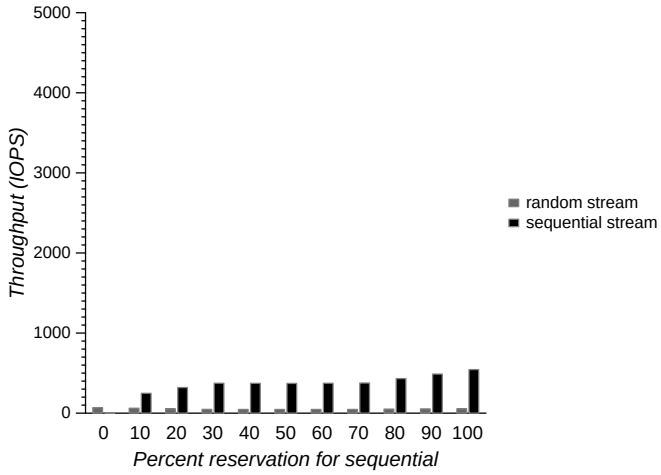
Each pair of bars in these graphs, showing the performance of one random and one sequential stream, represents the result of a single experiment. Always reserving a total of 100% of the reservable disk resources, we vary the reservation of the sequential stream from 0% to 100% and of the random stream from 100% to 0%. The ideal result for these sets of experiments would be a diagonal split of the bars, showing that each stream received exactly its reserved share.

We expected that utilization would allow for good control *if* the scheduler could actually control utilization for a hard disk. Figure 4(a) shows that this was the case. The results are very nearly a perfect diagonal split, representing nearly perfect control over the sharing of the disk performance. The slight discrepancies at high sequential reservations occur because Linux was not able to run the workload generator fast enough to keep the queue of sequential requests from emptying, occasionally allowing the scheduler to schedule extra requests from the other virtual device. Figure 4(b) shows the throughput that results from these allocations. As expected (and desired), the sequential request stream achieves significantly higher throughput, reflecting both more efficient use of its reserved time and isolation from the random request stream.

By comparison, throughput reservations are fulfillable, but provide poor control over the division of performance and lower overall performance. Figure 5(a) shows the results for the same experiment, but using throughput instead of utilization. The maximum reservable throughput was determined by sending a stream of random I/O requests to the drive and measuring the rate (which is somewhat better than the actual worst-case performance.) Each virtual



(a) Percent of nominal throughput



(b) Throughput, showing sequential I/O requests per second and I/O runs per second

Figure 5: Controlling performance using throughput: throughput received by one random and one sequential request stream from the throughput-based scheduler as the reservation of the sequential stream varies from 0% to 100% of the nominal throughput (and the reservation of the random stream varies from 100% to 0%).

device then reserved some fraction of that nominal throughput.

Figure 5(a) shows that each request stream received at least the throughput it reserved. However, the sequential stream always obtained an I/O rate higher than the reservable maximum when it had at least 10% reservation, because the sequential I/O requests took so little time to execute compared to the nominal case used to define the reservable I/O rate. The results are not at all like the nearly perfect diagonal split for the utilization-based driver and demonstrate very poor control over the sharing of the disk performance.

In practice, the throughput-based scheduler allows only a small fraction of the I/O rate to be controlled by reservation; the rest is determined by the scheduler's slack management policy. A sequential stream reserving 10% or more received far more than its reserved throughput—more, in fact, than the total reservable throughput. After fulfilling the reservation of the virtual device serving random requests, the scheduler was able to schedule several of the more efficient sequential runs using slack. For sequential reservations of 30% and above, our fair-sharing slack management policy effectively determined the performance achieved by both devices. Increasing the sequential reservation did not result in an increasing share of the achieved throughput. After fulfilling the reservation of the sequential stream, our fair sharing slack management policy scheduled some of the less efficient random requests. Thus, counter-intuitively, a sequential stream on a virtual device that had reserved 100% of the reservable throughput did not receive 100% of the actual throughput.

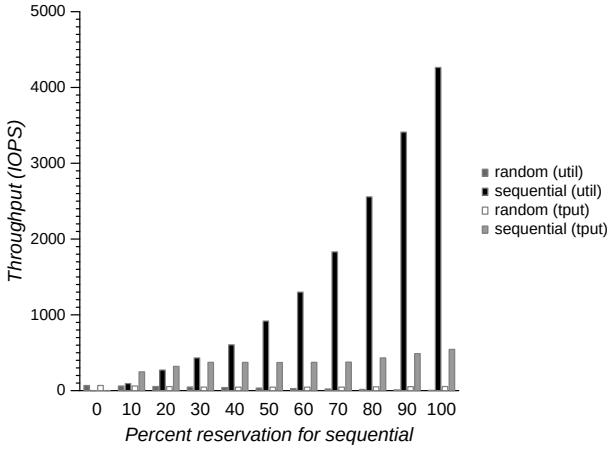


Figure 6: Efficiency comparison: throughput received by one random and one sequential request stream from the utilization- and throughput-based schedulers as the reservation of the sequential stream varies from 0% to 100% (and the reservation of the random stream varies from 100% to 0%) of utilization or throughput (as appropriate for the scheduler).

In addition to providing better control over the division of resources, the utilization-based scheduler provides higher overall performance. Figure 6 compares the observed throughput received by the request streams in the preceding experiments. Under the utilization-based scheduler, the performance of the sequential stream approached the maximum achievable throughput as its reservation increased. Under the throughput-based scheduler, it was capped by the reservation, which was in turn capped by the

maximum guaranteeable throughput.

4.3 Sensitivity to model errors

Because disk requests are not preemptible, the scheduler must use a model of the disk's performance when making scheduling decisions. For generality and to minimize scheduling overhead the Zygaria schedulers avoid detailed modeling of the underlying disk and instead use very simple models. The utilization-based scheduler's model has three parameters: the estimated service time for "random" requests; the estimated service time for "sequential" requests; and the policy for classifying a request as sequential or random. The throughput-based scheduler, on the other hand, uses the nominal (maximum reservable) I/O rate as its model.

The results suggest that the utilization-based scheduler's behavior is stable over wide ranges of parameters and is insensitive to wrong estimates in service time. On the other hand, the throughput-based scheduler is quite sensitive to errors in its model.

Utilization—random service time. The utilization-based scheduler is relatively insensitive to underestimates in the estimated random I/O service time, as shown in Figure 7. An underestimate corresponds to overestimating the potential throughput of the disk. The graph compares estimates of 20 msec (the default value), 5 msec, and 300 μsec (the default estimate for sequential requests). The results suggest that as long as there is a significant difference between the sequential and random estimates, the scheduler will behave correctly.

Utilization—sequential service time. Figure 8(a) shows that each request stream received the utilization it reserved, regardless of overestimates in the sequential request service time. The sequential stream receives less throughput with increasing overestimates, shown in Figure 8(b), because the scheduler dispatches fewer sequential requests at any given time to the disk.

Throughput-based scheduler. The throughput-based scheduler is very sensitive to errors in the estimated reservable throughput. The measured throughput of the disk we were using was 68 IOPS. We ran a set of experiments in which we overestimated the reservable throughput of the disk by 10% (75 IOPS) and 50% (102 IOPS). The results in Figure 9 show that the scheduler's behavior varies widely. For any overestimate in reservable throughput the random stream does not achieve 100% of its reservation—as expected, since the disk is not capable of that many random I/O requests per second.

4.4 Short-term behavior

The previous results confirmed that using utilization as a metric for I/O scheduling provides better long term control than using throughput. However, short term time-varying behavior has to be considered as well. Here we consider two key aspects: that the scheduler gives multiple virtual devices their proper utilization when the workload is stable, and that it responds properly when the workload changes.

Figure 1 (§1) shows the cumulative disk service time that two request streams received over time. We picked one single experiment from the set used to generate Figure 4, where one stream reserved 70% of the disk's utilization and sent sequential requests, while the other reserved 30% and sent random requests. For this experiment we sample the achieved utilization every 500ms. After a small initial startup transient caused by the workload generators starting at slightly different times, both streams get their proper amount of service time. This analysis of transient performance confirms that our approach provides more than just statistical guarantees, fulfilling reservations as a fraction of each disk-second.

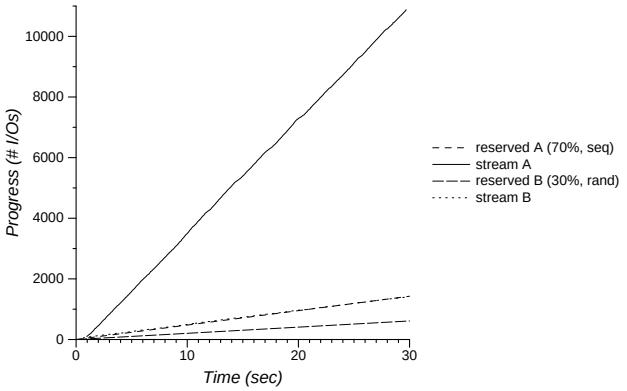
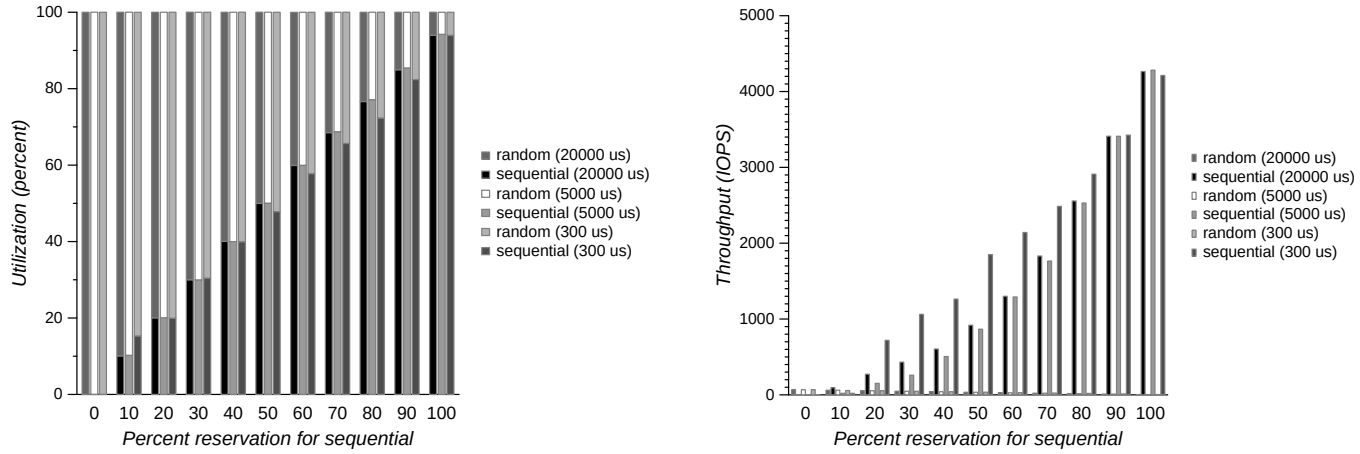


Figure 10: Time series of cumulative throughput received by two request streams, one random with a 30% reservation and one sequential with a 70% reservation.

Figure 10 shows cumulative I/Os for the *same* workload, but using the throughput based scheduler. Accordingly, we picked one single experiment from the set used to generate Figure 5, with a 70% reservation for the sequential stream and a 30% reservation for the random stream, sampling the progress in terms of number of I/O's also every 500ms. Each stream achieves a much higher throughput than its reservation because the reservable throughput is necessarily based on worst-case assumptions which are inaccurate for the sequential stream (A). After fulfilling the throughput reservations it is up to the scheduler how to handle slack, which does not have to be distributed according to the reservations. This is a problem inherent in using a workload-dependent metric (*e.g.*, throughput) to reserve storage performance, and is not a problem with the scheduler.

Figure 11 shows a more challenging scenario, with three request streams becoming active at different times. Stream 1 is active for the duration of the experiment, the others only part of the time. Stream 2 serves sequential requests, while the others serve random requests. All three have established their reservation at the beginning of the



(a) Utilization

(b) Throughput, with sequential (0, 0) results omitted

Figure 7: Sensitivity of the utilization-based scheduler to errors in the random I/O service time estimate. Performance received by two request streams, one random and the other sequential, from the utilization-based scheduler with increasingly underestimated random I/O request service time (in parentheses).

run. (This is a variation of an experiment used to evaluate the throughput-based scheduler, as reported in our earlier work [27].)

The time series shows three key effects. First, each stream receives at least the performance reserved by its device or fair share after transients due to workload changes. Second, when a stream begins sending requests, it briefly receives more than its usual performance; for example, stream 2 at $t = 10$ sec. This occurs because the scheduler uses token buckets to track a stream's recent utilization. When a stream is inactive its bucket will accumulate tokens (up to one second worth of its reservation), which are then available immediately when it starts sending requests. Finally, when a stream stops sending requests there is another short transient where a stream that has been getting less utilization will briefly get more; for example, stream 1 at $t = 30$ sec. This occurs because there is slack available from the stopped stream and sharing is determined by the scheduler's slack management policy. We used a fair sharing policy based on a moving average over a window of a few seconds. Both transient effects are due to using measurements that estimate utilization based on recent behavior, which introduces a lag in response compared to an oracular instantaneous measure. This is a general problem in feedback control systems.

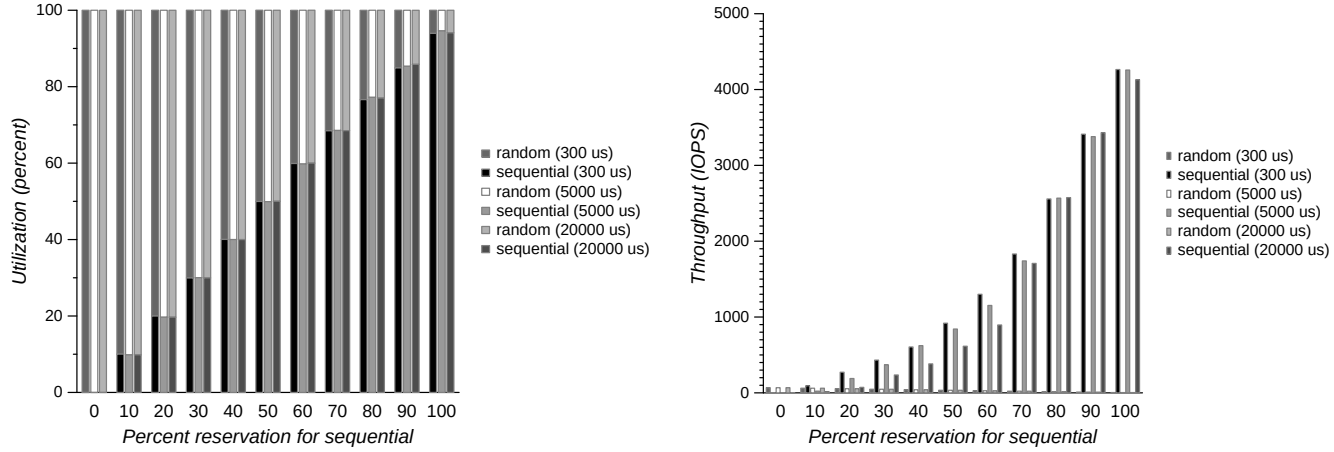
5 Related work

Storage virtualization. Storage virtualization has 2 dimensions: capacity and performance. Many commercial products virtualize storage capacity [12, 9]. The virtualization of storage performance has been the subject of many research projects. Stonehenge [11] tries to address both, but because it uses bandwidth as its metric of storage performance, its reservable performance is limited to a fraction of the raw disk performance. Parallax [25] was designed to manage the storage requirements of large numbers of virtual machine images. These images include system data, like swap files, which are particularly crucial for performance. While optimizing overall storage performance, Parallax cannot guarantee storage performance for individual virtual machines. Storage Tank [17] employs data placement on the storage devices as a mechanism for storage virtualization. SLEDS [6] and Façade [14] achieve statistical performance guarantees, by dynamically adjusting the share provided to each application stream.

The above approaches address virtualization in the context of large-scale distributed storage systems. Argon [22] and Fahrrad [18] were designed for individual storage servers. Following the idea of virtual disks, Argon attempts to provide each application stream with at least a configured fraction of the throughput achieved when the stream had the server to itself. Fahrrad goes a step further, providing hard performance guarantees in part by fully accounting for all interference between request streams.

Application requirements and behavior. Each application has individual requirements for storage performance and different applications may use incommensurable metrics for specifying those requirements. At the same time, the resulting performance depends on the application's behavior, in particular the offered workload. There are three different approaches for managing storage performance based on application requirements and behavior:

First, application requirements can be used to assign applications to specific storage devices, e.g. LUNs or disk arrays [12, 9]. However, these systems cannot (directly)



(a) Utilization

(b) Throughput, with sequential (0, 0) results omitted

Figure 8: Sensitivity of the utilization-based scheduler to errors in the sequential I/O service time estimate. Performance received by two request streams, one random and the other sequential, from the utilization-based scheduler with increasingly overestimated sequential I/O request service time (in parentheses).

control storage performance of single devices, hence this approach inevitably results in over-provisioning the system. The Minerva system [1, 2] refines this approach by considering short-term application behavior when assigning devices to workloads. But again, this scheduling mechanism does not control the device directly to affect (current) performance.

Second, Façade [14] and SLEDS [6] control request latency by dynamically changing the share of storage performance given to each application. Although this mechanism can compensate for changes in application behavior, it makes it difficult or impossible to determine the admissibility of a set of applications and their requirements.

Third, disk I/O schedulers tailored for multimedia applications directly use application metrics to control I/O scheduling. Since these schedulers are built assuming multimedia-specific application behavior, *e.g.*, periodicity, they are not directly applicable to other domains.

Scheduling and Metrics. Many research groups have developed I/O scheduling algorithms. The historical focus was on overall efficiency [16, 20], while later algorithms provided for control performance using various metrics. Chen and Patterson [7] discuss the most common metrics, throughput (similar to bandwidth and I/O rate) and response time (similar to latency), including their tradeoffs.

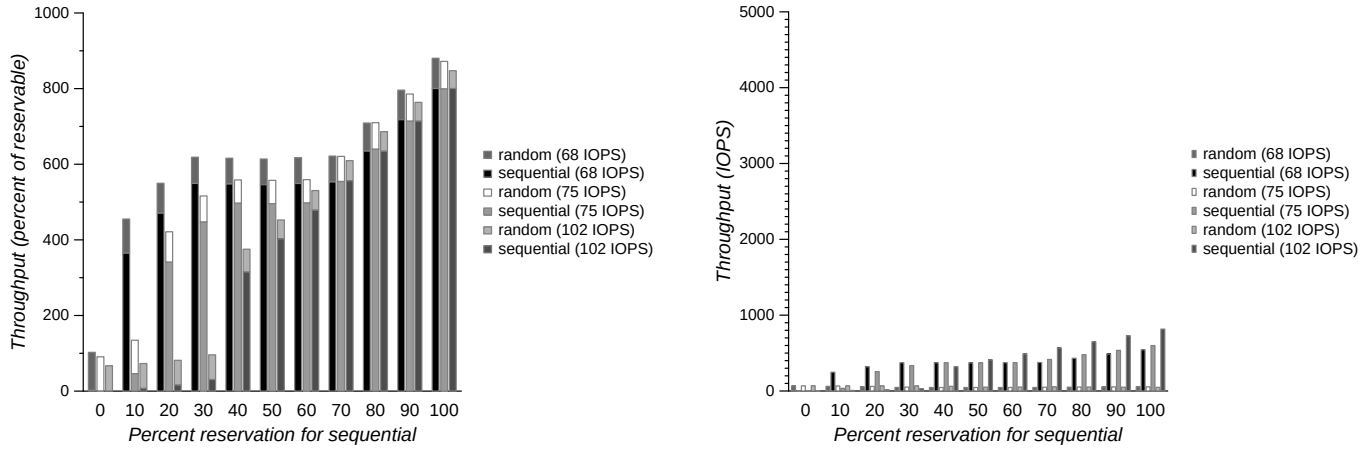
Bandwidth and I/O rate are the most common metrics, as seen originally in XFS [8]. The original Zygaria driver [27] allows sessions to reserve an I/O rate based on worst-case execution time estimates and enforces those reservations using EDF scheduling. As discussed earlier, the worst-case execution time assumption can be off by orders of magnitude when a workload exhibits better than worst-case locality. Other schedulers focus on sharing of bandwidth or I/O rate, including lottery scheduling [23], hierarchical disk sharing [28], and various fair queuing algorithms [10, 4, 13]—generally without the ability to guarantee rates.

The DAS scheduler in DROPS [19] supports hard real-time, soft real-time and best-effort applications. DAS allows arbitrary reservation granularity in terms of throughput. It tries to optimize disk utilization by dividing a job into mandatory and optional parts; mandatory requests are guaranteed and optional requests are executed if there is slack left from mandatory requests. Since reservations have to be made using worst-case assumptions, reservable throughput is low.

Latency is another common metric. Schedulers such as Façade [14] and SLEDS [6] exploit the tradeoff between latency and throughput: longer queue lengths produce longer latencies, but allow more efficient disk head movement. They typically use an active feedback control system to adjust the queue length or share given to each session to maintain latency goals.

Specialized multimedia schedulers directly support traditional real-time metrics, *e.g.*, periods or deadlines. They range from *simple* EDF schedulers, *e.g.*, Clockwise [3] to multi-level hierarchical schedulers, *e.g.*, Cello [21], MARS [5], and the work by Wijayaratne and Reddy [26], supporting various classes of traffic.

Comparing time- and bandwidth-based resource allocation, Cello concludes that time-based allocation is more suitable for real-time workloads (*e.g.*, video) and recommends bandwidth allocation for other, more general workloads (*e.g.*, file-servers).



(a) Percent of nominal throughput

(b) Throughput, with sequential (0, 0) results for 75 and 102 IOPS omitted

Figure 9: Sensitivity of the throughput-based scheduler to errors in nominal throughput estimate. Performance received by two request streams, one serving random and one sequential requests, from the throughput-based scheduler with increasingly overestimated nominal throughput.

Isolation. Isolating one I/O workload from another has proven to be inherently difficult, hence relatively few disk schedulers address isolation. Argon [22] provides *insulation* between request streams by limiting and mitigating their interference. Using weighted fair sharing of a disk among multiple request streams, Argon provides soft bounds on the overhead due to sharing but does not provide any other performance guarantees. e.g. feasibility—which can change due to non consistent workloads. Fahrrad [18] provides complete isolation between request streams by explicitly accounting for all seeks, both within and between streams, and charging each stream for the seeks it causes.

6 Conclusions

Many applications require performance guarantees from the storage subsystem. Virtual storage devices can both guarantee the required performance and eliminate interference from competing workloads. Bandwidth, the most common way to express, measure, and manage storage performance, depends upon application behavior and therefore requires worst-case assumptions about I/O patterns. This leads to low efficiency and little actual control over the apportioning of the disk performance and thus bandwidth is a poor basis for performance reservation.

Disk time utilization is a more effective basis for virtualizing disk performance. It is 100% reservable and easily manageable, hence achieved performance closely matches reserved performance. By embedding knowledge of application I/O patterns in the reservations and isolating workloads, utilization-based virtual disks can provide greater throughput than throughput-based schedulers. The results from our utilization-based scheduler demonstrate these effects, showing that utilization-based disk virtualization is both feasible and effective.

Acknowledgments

This work was supported in part by the National Science Foundation Award No. CCF–0621534.

References

[1] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. on Comp. Sys.*, 19(4):483–518, Nov. 2001.

[2] E. Borowsky, R. Golding, P. Jacobson, A. Merchant, L. Schreier, M. Spasojevic, and J. Wilkes. Capacity planning with phased workloads. In *Proc. 1st Intl. Workshop on Software and Performance*, pages 199–207, Oct. 1998.

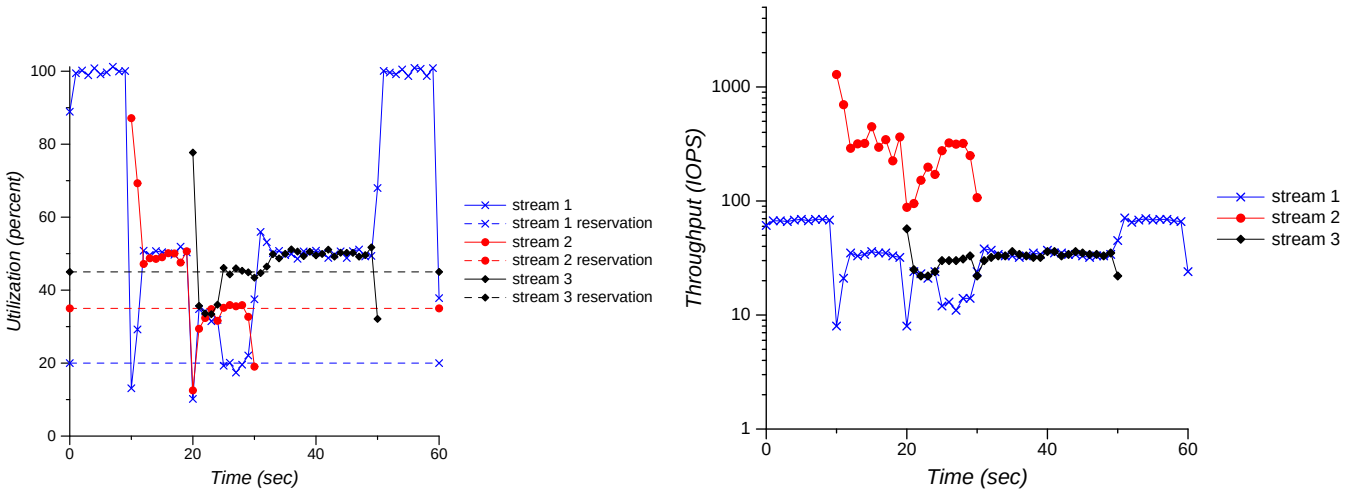
[3] P. Bosch, S. J. Mullender, and P. G. Jansen. Clockwise: A mixed-media file system. In *Proceedings of the 1999 IEEE International Conference on Multimedia Computing and Systems (ICMCS '99)*, pages 277–281, June 1999.

[4] J. L. Bruno, J. C. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *Proceedings of the 1999 IEEE International Conference on Multimedia Computing and Systems (ICMCS '99)*, pages 400–405, 1999.

[5] M. M. Buddhikot, X. J. Chen, D. Wu, and G. M. Parulkar. Enhancements to 4.4 BSD UNIX for efficient networked multimedia in project MARS. In *Proc. of the 1998 IEEE Intl. Conf. on Multimedia Computing and Systems*, pages 326–337, June 1998.

[6] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance virtualization for large-scale storage systems. In *Proc. of the 22nd Symp. on Reliable Distributed Systems*, pages 109–118. IEEE, Oct. 2003.

[7] P. Chen and D. Patterson. Storage performance—metrics and benchmarks. *Proceedings of the IEEE*, 81(8):1151–1165, Aug. 1993.



(a) Utilization, sampled in half-second intervals

(b) Throughput, sampled in half-second intervals

Figure 11: Time series for the performance received by three streams under the utilization-based scheduler. The (start, stop) times are (0, 60) for random stream 1, (10, 30) for sequential stream 2, and (20, 50) for random stream 3.

[8] S. Ellis and J. Raithel. Getting started with XFS filesystems. Document Number 007-2549-001, SGI, Inc., Mountain View, CA 94043, 1994.

[9] EMC. EMC ControlCenter software family data sheet. http://www.emc.com/products/storage_management/controlcenter/pdf/H1082_%CC_Stor_Fam_LDV.pdf, 2004.

[10] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. In *Proc. of SIGCOMM 1996, the ACM Symp. on Communications, Architectures, and Protocols*, pages 157–168, 1996.

[11] L. Huang, G. Peng, and T. cker Chiueh. Multi-dimensional storage virtualization. *SIGMETRICS Perform. Eval. Rev.*, 32(1):14–24, 2004.

[12] IBM Corp. IBM TotalStorage Productivity Center. http://www.ibm.com/servers/storage/software/center/index.html, 2004.

[13] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proc. of SIGMETRICS 2004, the Intl. Conf. on Measurement and Modeling of Computing Systems*, pages 37–48. ACM SIGMETRICS, June 2004.

[14] C. R. Lumb, A. Merchant, and G. A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, Apr. 2003.

[15] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002.

[16] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, Aug. 1984.

[17] J. Menon, D. A. Pease, R. M. Rees, L. Duyanovich, and B. Hillsberg. Ibm storage tank - a heterogeneous scalable san file system. *IBM Systems Journal*, 42(2):250–267, 2003.

[18] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. In *Eurosys 2008*, April 2008.

[19] L. Reuther and M. Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, Dec. 2003.

[20] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX Technical Conference*, pages 313–323, Jan. 1990.

[21] P. J. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of the 1998 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 44–55, Madison, WI, 1998.

[22] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies (FAST)*, pages 5–5, 2007.

[23] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI'94)*, Nov. 1994.

[24] M. Wang. Performance modeling of storage devices using machine learning. *PhD Thesis, CMU-CS-05-185*, January 2006.

[25] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: managing storage for a million machines. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 4–4, 2005.

[26] R. Wijayaratne and A. L. N. Reddy. Integrated QOS management for disk I/O. In *Proc. of the 1999 IEEE Intl. Conf. on Multimedia Computing and Systems*, pages 487–492, June 1999.

[27] T. M. Wong, R. Golding, C. Lin, and R. Becker-Szendy. Zygaria: Storage performance as a managed resource. In *12th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS06)*, Apr. 2006.

[28] J. Wu, S. Banachowski, and S. A. Brandt. Hierarchical disk scheduling for multimedia systerms and servers. In *Proceedings fo the ACM International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '05)*, pages 189–194, Stevenson, WA, June 2005. ACM.