Zygaria: Storage performance as a managed resource

Theodore M. Wong, Richard A. Golding, Caixue Lin[*] and Ralph A. Becker-Szendy
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120
theowong@us.ibm.com, rgolding@us.ibm.com, lcx@soe.ucsc.edu, ralphbsz@us.ibm.com

Abstract

Large-scale storage systems often hold data for multiple applications and users. A problem in such systems is isolating applications and users from each other to prevent their workloads from interacting in unexpected ways. Another is ensuring that each application receives an appropriate level of performance. As part of the solution to these problems, we have designed a hierarchical I/O scheduling algorithm to manage performance resources on an underlying storage device. Our algorithm uses a simple allocation abstraction: an application or user has a corresponding pool of throughput, and manages throughput within its pool by opening sessions. The algorithm ensures that each pool and session receives at least a reserve rate of throughput and caps usage at a limit rate, using hierarchical token buckets and EDF I/O scheduling. Once it has fulfilled the reserves of all active sessions and pools, it shares unused throughput fairly among active sessions and pools such that they tend to receive the same amount. It thus combines deadline scheduling with proportional-style resource sharing in a novel way. We assume that the device performs its own low-level head scheduling, rather than modeling the device in detail. Our implementation shows the correctness of our algorithm, imposes little overhead on the system, and achieves throughput nearly equal to that of an unmanaged device.

1. Introduction

Companies looking to reduce the high cost of storage often aggregate data onto shared virtualized storage systems, which can reduce the infrastructure and management overhead but can lead to unexpected interference between applications with potentially divergent performance requirements. For example, one user may be running a media player with deadlines when another user starts a storage-intensive file indexer. If the two users share some storage device, then the applications will compete with each other for performance resources, which may result in the media player missing deadlines. On a larger scale, a transaction-processing application may experience performance degradation when a backup process begins. In our experience, such competition is not a rare occurrence, and will likely become more frequent as these systems grow and as more applications share them.

The alternative to aggregation is to dedicate a storage device or logical unit to an application, which isolates applications but at the cost of complex manual configuration and inefficient resource utilization. Moreover, configurations are usually based on a snapshot of application behavior, and must be revisited as either the application requirements or the hardware infrastructure change.

A virtualized storage system must therefore provide assurances that the behavior of one application will not interfere with the performance of other applications. We are developing a storage system called *Kybos*[1] that manages the resources allocated to an application according to a specification of *reserves* and *limits*. A reserve specifies the amount of a resource whose availability Kybos will guarantee for the application. A limit restricts the additional amount of a resource that Kybos will provide to the application if unused resources exist. The limit can be used, for example, to ensure that housekeeping operations or backup do not use more than a certain amount of system performance, always leaving the remainder for regular applications.

Kybos provides virtualized storage in a distributed system that is built from many small, self-contained storage servers called *bricks*. Each brick enforces isolation locally between applications that share it. Internally, Kybos places data on bricks such that the system delivers reasonable overall performance, and reorganizes data in response to changes in the application behavior or the infrastructure.

Each brick in Kybos has the following goals for managing its performance resources:

- Reserve enforcement. An active application should get

[*]Computer Science Department, Unversity of California at Santa Cruz, Santa Cruz, CA 95064

[1]From the Greek $\kappa\upsilon\beta o\varsigma$ (cube).

at least its reserve amount on average from the brick, regardless of the behavior of any other applications.

- Limit enforcement. An application should receive at most its limit amount on average from the brick.
- Fair sharing of additional resources. Each active application should receive a fair share of any unused resources on a brick.

We have designed a brick-level *hierarchical I/O scheduling algorithm* to achieve these goals, and analyzed the performance results obtained from an implementation called *Zygaria*.[2] Zygaria is layered over a disk or a RAID device that performs its own low-level head scheduling. In addition to meeting the above goals, the I/O scheduler in Zygaria tries to keep the device busy with enough I/O requests to yield efficient head movement, and helps the device to take advantage of locality in the workload of an application by batching I/Os together. The scheduler only controls throughput over time intervals of one second or so, rather than providing hard real-time guarantees. Thus, the scheduler can treat the underlying device essentially as a black box, unlike those that model devices in detail.

The Zygaria I/O scheduler is novel in combining reserve and limit enforcement on I/O performance resource usage with fair sharing of best-effort resources. It uses token buckets [21] to track how close an application is to its limit and how far it is operating below its reserve. It also maintains a moving average measurement recent performance that is used to determine how best to distribute any unused resources.

Our experimental results in §5 confirm that our algorithm enforces reserves and limits, and thus isolates the workloads of competing applications. They also demonstrate how otherwise unused resources are shared fairly between active applications, and that the advantages of managed performance come at low CPU and throughput costs.

2. System model

We have designed Zygaria to be layered over a disk or a RAID device. We assume that the device performs its own low-level head scheduling, instead of modeling the device in detail. We also assume that the device:

- Has a known worst-case throughput. We measure the throughput with a stream of small, random I/Os.
- Has an approximately constant average seek time.
- Is more efficient if it has a queue of several outstanding I/Os. Efficiency should increase rapidly when going from one I/Os to a few, and more slowly thereafter.

[2]From the Greek $\zeta\upsilon\gamma\alpha\rho\iota\alpha$ (balance; scales).

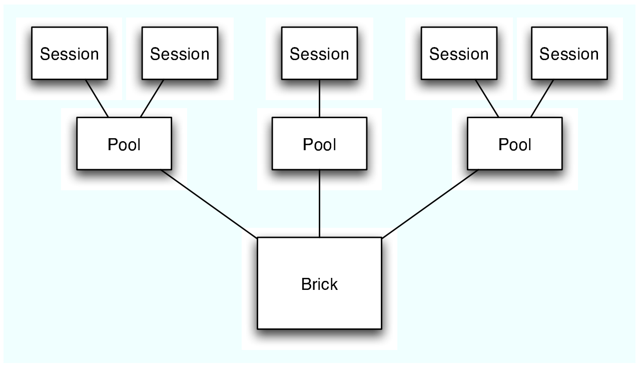


Figure 1. Relationship between sessions, pools, and a brick.

- Has a starvation-free driver.
- Executes a sequential run of I/Os nearly as fast as the first I/O in the run, up to a reasonable run length.

These assumptions are reasonable for modern disks [17].

3. Hierarchical resource allocation

Zygaria models performance resource allocation policies as a hierarchical arrangement of *pools* and *sessions* (Figure 1). A pool is a long-term entity that an administrator creates to manage the I/Os for an application. The administrator controls the amount of resources allocated for a pool. A session is a short-term entity that an application process creates from within a pool to manage one stream of I/Os. Processes can create an arbitrary number of sessions provided that the total amount of resources allocated for the sessions does not exceed the amount allocated for the pool. For example, the administrator might configure the pool for a media server application to support ten concurrent media streams across all the files in the library of the server. Up to ten media player processes could then open sessions to play one media stream each.

Each pool or session specifies a {reserve, limit} pair of requirements on their average received I/O rate, where the limit is greater than or equal to the reserve. Zygaria guarantees that an application can execute I/Os at a rate up to its reserve, and allows the application to execute at a rate up to the limit when unused resources are available. These resources may be unreserved, be from other pools or sessions that are operating below their reserve, or be from recent I/Os that execute more efficiently than expected. The reserve may be zero, meaning that all I/Os are best-effort, while the limit may be infinite.

Pools and sessions specify their requirements in terms of I/O runs per second, rather than I/Os per second or bytes per second. An I/O run is a set of sequential I/Os, up to a fixed maximum amount of data. We expect each run to require a disk head seek and rotation, thus a runs per second specification is a rough proxy for the disk utilization that a given requirement implies.

Feasible reserve and limit values for a pool or session depend on the resources available from the underlying device. The pools in the system are feasible if the sum of their reserves does not exceed the worst-case throughput of the device, thus defining the admission criterion for pools. Similarly, the sessions in a pool are feasible if the sum of their reserves does not exceed the reserve of a pool. Limit values are arbitrary, but Zygaria ensures that any session will never get more than the limit of its pool.

4. I/O scheduling algorithm

Our hierarchical I/O scheduling algorithm ensures that sessions and pools receive their reserve I/O rates on average, and that they receive no more than their limit I/O rates, by combining the characteristics of an earliest-deadline-first (EDF) algorithm [15] with those of slack-stealing algorithms for CPU schedulers [14, 25, 2] and proportional-share schedulers [29]. To accomplish these objectives, the scheduler computes for each I/O a *release time*, which is the time after which the I/O can be executed without its session and pool exceeding their limits, and a *deadline*, which is the time by which the I/O must be executed for its session and pool to receive their reserves. The release time can never be later than the deadline, given that the limit is never lower than the reserve (§3). Release time and deadline are computed using a token bucket, as described in §4.1. If the scheduler finds that the deadline of the I/O with the earliest deadline has expired, it sends that I/O to the underlying device for execution. Otherwise, it takes advantage of the implicit slack in the schedule to execute other I/Os, selecting I/Os such that the unused resources are shared fairly among sessions and pools (§4.2).

Figure 2 shows the scheduler data structures; note how they mirror the session and pool architecture in Figure 1.

4.1. Limits and reserves

Our algorithm maintains two token buckets for each session and pool to ensure that they are staying within their limits and reserves.

The reserve bucket measures how much a session or pool is operating *below* its reserve. It has a refill rate r equal to the reserve rate, or zero if there is no reserve. An I/O must run as soon as possible after the reserve bucket accumulates a token for the I/O, which sets the deadline at $(n-n_r)/r$ seconds into the future; n is the number of tokens the I/O requires, and n_r is the number of tokens currently in the reserve bucket.

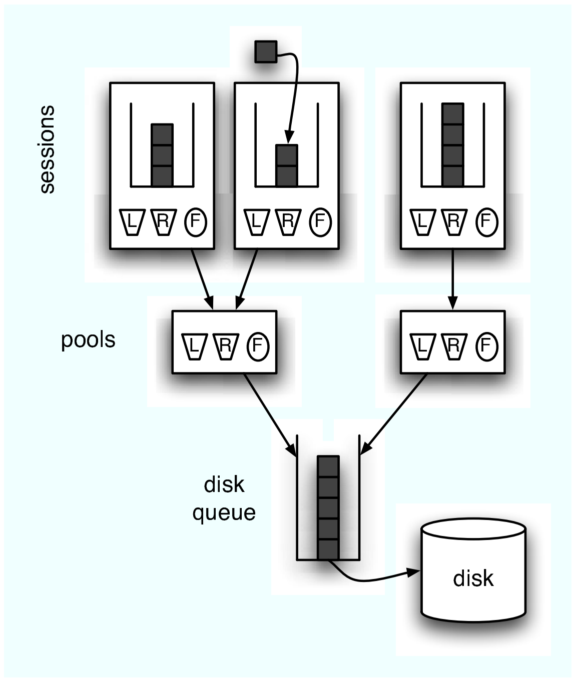


Figure 2. Token bucket hierarchy and I/O queue structure for sessions, pools, and the underlying device. (R) and (L) are reserve and limit token buckets; (F) is a moving average estimator used for fair sharing.

The limit bucket tracks how close a session or pool is to its limit. It has a refill rate l equal to the limit rate, or $+\infty$ if there is no limit. An I/O must wait until the limit bucket has accumulated a token for the I/O, which sets the release time at $(n-n_l)/l$ seconds into the future; n is the number of tokens the I/O requires, and n_l is the number of tokens currently in the reserve bucket.

4.2. Fair sharing

Our algorithm supports a model of resource sharing that we call *water-level fair sharing*. Once all active sessions and pools have received their reserve, they will receive additional best-effort resources such that they will tend to obtain the same I/O rate, subject to any limit constraints. Water-level sharing behaves as if the extra resources were being poured into the pools, and thus will tend to give performance to the lowest pools until all pools get the same amount. Similar resource sharing occurs among sessions within a pool.

Our algorithm maintains a moving average of the recent performance of each session and pool. The current implementation keeps a window over the last five seconds, in 20 buckets that are a quarter-second in width. The recent throughput of I/Os is estimated by taking a weighted average over the buckets: $T = \sum_{i=0}^{19} b_i \cdot \alpha^i$, where b_0 is the throughput of the current quarter-second period, and α is a decay factor. To understand why we chose this unusual weighted average over a traditional non-weighted average, we consider the algorithm to be a control system with negative feedback when it makes a fair sharing decision: the input signal (the throughput received by a session or pool) is filtered with an averaging function, and then used in a feedback function. In our algorithm, the feedback function operates by comparing the throughput from different sessions or pools. A traditional moving average has a "top-hat" shape as the impulse response, which Fourier transforms into the spherical Bessel function $j_o(\omega) = \text{sinc}(\omega)$ in frequency space. Unfortunately, the first negative minimum of $j_0(\omega)$ comes close to violating the Bode criterion [18], which causes the control system to react to workload transients by oscillating for extended periods. The Bode diagram of the exponential-weighted average we have chosen has a very large phase margin, causing the fair sharing control system to stabilize.

If the algorithm determines that all releasable I/Os have deadlines in the future, it takes advantage of the implicit slack in the I/O schedule to schedule additional I/Os. It finds the pool with the lowest moving average, finds the session with the lowest moving average in that pool, and schedules an I/O from that session. This process moves the system toward water level.

When an inactive session becomes active, the combination of a five-second moving average with the approach of always picking the pool and session with the lowest average means that the session will preferentially get extra performance until it has caught up with other sessions. However, the algorithm dampens this transient effect quickly.

Our decision to implement water-level sharing is orthogonal to the other design decisions made for our scheduling algorithm. One could implement other definitions of fair sharing—such as proportional shares or equal increments over reserve—by replacing the algorithm described here with other algorithms to choose the pools and sessions that should receive unused resources. For example, we have begun (but not yet completed) an investigation into using a lottery algorithm [24, 26] to select the I/O to schedule, on the hypothesis that this would help smooth transient behavior.

We demonstrate water-level fair sharing behavior in the experiments discussed in §5.1.

4.3. I/O scheduler

The scheduler in Zygaria takes I/Os for different sessions and determines when to send them to the underlying device for execution, as shown in Figure 2. The scheduler runs each time an I/O arrives or completes, and when the release time or deadline passes for an I/O queued in any session.

The scheduler restricts the number of I/Os outstanding at the device to balance device efficiency with accurate scheduling. Disk head schedulers in particular are more efficient if they have more I/Os to choose from, especially if they can process multiple adjacent I/Os without a head seek. However, our scheduler has no control over I/Os once it sends them to the device, thus if it sent several I/Os before their deadline, and an I/O with a short deadline subsequently arrived, the new I/O might be delayed long past its deadline.

Each time the scheduler runs, it enters a loop to schedule as many I/Os as it can. For each iteration, the scheduler picks one I/O to send to the device queue. When the scheduler sends an I/O to the device, it updates all of the token buckets and moving average statistics for the pool and session of the I/O. The scheduler stops either when no more I/Os are past their release time or when too many I/Os are outstanding at the device. It then arranges to wake up at the earliest deadline or release time of any I/O queued in any session, if there is one.

The scheduler begins a loop iteration by identifying releasable I/Os. For each session with a non-empty queue, the scheduler computes two release times for the I/O at the head of the queue: the time given by the limit bucket of the session, and the time given by the limit bucket of the pool of the session. If either of these times is in the future for an I/O, the scheduler excludes the I/O from further consideration, ensuring that no session or pool exceeds its limit.

The scheduler next runs a modified EDF algorithm to select an I/O whose deadline has expired. For each session with a releasable I/O, the scheduler assigns a deadline to the I/O that is the earlier of the deadline given by the reserve bucket of the session, and the deadline given by the reserve bucket of the pool of the session; by assigning the earlier of the two deadlines, it ensures that both the session and the pool will receive their reserves. The scheduler selects the I/O with the earliest deadline, and queues the I/O at the disk if the deadline has expired; note that waiting until deadlines have expired does not affect the average throughput of an I/O stream.

When running the modified EDF algorithm, the scheduler tries to send batches of I/Os rather than single I/Os to help the device to take advantage of locality in the I/O stream of a session. To achieve this, the scheduler treats a run of up to 32 KB of sequential I/Os in a session queue as a single I/O, counting them as a single I/O run for token buckets and moving averages. It also batches I/Os from one session and sends them to the device together. The size of the batch is limited to the maximum of the number of releasable I/Os in the session queue and the session reserve rate r. The scheduler will thus batch at most one second of I/Os at the reserve rate, which can increase device efficiency but can also increase the variability of service time—but only for sessions that have many I/Os in flight and are thus likely to be throughput- rather than latency-sensitive.

If the scheduler determines that all releasable I/Os have deadlines in the future, then it can take advantage of implicit slack in the schedule to insert additional I/Os ahead of those with deadlines. To accomplish this, it selects I/Os that will achieve water-level fair sharing (§4.2).

5. Results

We have evaluated Zygaria for correctness and efficiency. We found that in the long term it provided reserves, enforced limits, and shared fairly, and that in the short term it behaved well as I/O streams started and stopped. We also found that the benefits of managed performance came at low cost: CPU usage was low, and throughput was nearly as good as running without Zygaria.

Zygaria is a loadable block device driver for the Linux 2.6.11 kernel, in about 2200 lines of commented C code. The driver sits above a disk device, and exports a set of block devices named `/dev/zygaria`*`PS`*, where *`P`* is the pool and *`S`* is the session. The pools and sessions all share the underlying device. A user-level program sets reserves and limits on the pools and sessions via an `ioctl()` call.

Zygaria does not supply or use a cache, and does not sort its queues in elevator order. In our experiments, we restricted the number of I/Os that Zygaria had outstanding at the underlying device to 10. We configured the underlying device driver to use the Linux 2.6 “anticipatory” disk scheduler, but disabled its support for deadline scheduling and anticipatory scheduling, thus turning it into a traditional one-way elevator scheduler.

We collected all the experimental data on an IBM TotalStorage NAS100 server—a 1U rack-mount server, with a 1.2 GHz Pentium III processor, 512 MB memory, and four IBM IC35L120 DeskStar disks (120 GB, 7200 RPM, separate IDE interfaces). We used one of these disks for Zygaria, and a second disk for the root file system.

We found that the disks performed 112 IOPS for a random workload of 1024-byte reads over the whole disk, with 10 I/Os outstanding at the disk at a time. We fixed this rate as the reservable throughput of the disk, and in some experiments we report results as a percentage of this reservable throughput.

For all experiments, we set the decay factor α for the fair sharing moving average filter (§4.2) to 0.9.

A simple user-level workload generator created a variety of I/O patterns, including sequential (starting at arbitrary offsets) and random (within a range of disk addresses). In most experiments, the generator used a closed arrival process with zero think time and a fixed number of I/Os outstanding at any time. For other experiments, the generator used an open arrival process with a fixed inter-arrival time to simulate periodic traffic; since the generator had only one I/O outstanding at a time, an I/O that could not be scheduled at the desired time executed as soon as possible afterward.

The generator also recorded a trace of the I/Os it executed. With cooperation from the Zygaria driver, the trace contained information on when an I/O arrived and completed, and when it passed various control points in the driver. We used these traces to generate the time-series graphs in the following sections.

We report 95% confidence intervals for several experiments. We used a batch-means analysis to determine the confidence interval width.

5.1. Correctness

The first experiment showed that the algorithm handles reserves and limits properly, and shares throughput fairly. Recall that the scheduler provides water-level fair sharing, which behaves as if extra throughput were being poured into sessions, tending to give throughput to the lowest sessions until all sessions get the same amount.

The experiment measured the amount of throughput that each of several sessions got as the system was able to process more I/Os. This happened because some of the sessions caused the disk head to seek less than in the worst case that fixed the reservable throughput. This experiment had three sessions, all in the same pool. The experiment varied the range of disk addresses accessed by some sessions:

Session	Reserved	Limit	Range
1	15%	40%	full disk
2	35%	none	fraction
3	50%	none	fraction

Each session read data at uniformly random offsets. Session 1 distributed I/Os over the entire disk, while the other two sessions distributed I/Os over only a fraction of the disk. The reserve and limit values are expressed as a percentage of the reservable throughput of 112 IOPS.

Figure 3 shows that the scheduler implemented reserves, limits, and sharing properly as sessions 2 and 3 accessed smaller fractions of the disk, which decreased the seek distance for the disk head. Point 1/1 is the least efficient, at which all I/Os ranged over the full disk; all sessions received their reserve or slightly more. As sessions 2 and 3 accessed smaller fractions of the disk, head movement became more efficient, and the scheduler could admit more best-effort I/O. Water-level sharing worked to give all sessions the same throughput, and ensured that the session with lowest throughput received the benefit as more I/Os can be scheduled. This happened even though the I/Os in session 1

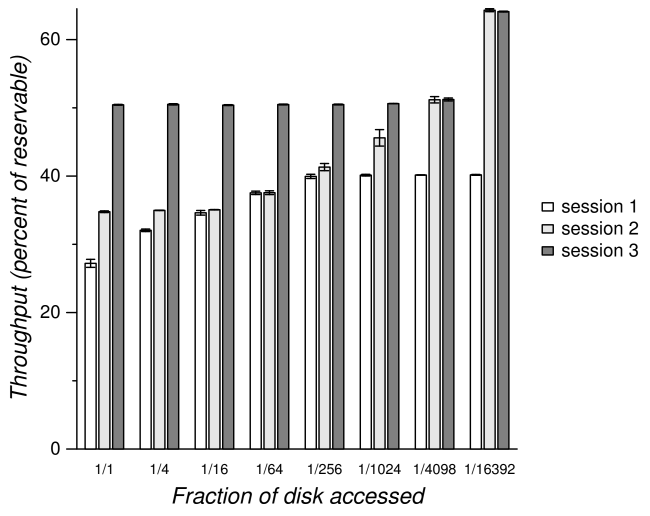


Figure 3. Fair sharing among streams in one pool. Experiment varied the disk address range that two of the streams accessed to the fraction of the disk shown on the x-axis, increasing efficiency and making increased throughput possible. Results are the average of 10 runs; bars show the 95% confidence interval.

could not be processed as efficiently as those in other sessions. Session 1 was also capped when it reached its limit.

We performed a similar experiment where the sessions were spread across multiple pools. Those results, omitted here for brevity, showed that the scheduler enforced both reserves and limits of both pool and session. They also showed that sharing works correctly in the hierarchical system: pools first shared any best-effort throughput from the device, and sessions then shared any best-effort throughput from within their pool.

5.2. Time-varying behavior

The next experiment illustrated how Zygaria behaved in the short term as the offered workload changed. Figure 4 shows an example of a 60-second run with three sessions:

Session	Reserved	On at	Off at
1	10% / 11 IOPS	0 s	60 s
2	20% / 22 IOPS	10 s	30 s
3	40% / 45 IOPS	20 s	50 s

Each session was driven by a generator with a closed arrival process with zero think time that maintained 20 I/Os outstanding, which used up as much throughput as the scheduler would give it.

The experiment showed how resources were shared by multiple sessions. A solo active session received the entire throughput of the disk. Two active sessions shared the throughput equally. In the period from 20–30 seconds, there were three active sessions, but session 3 had a high reserve. Session 3 received its reserve while the other two sessions shared the remaining throughput equally.

Figure 4(a) shows that changes in the offered workloads caused only short transient effects. For example, at 10 seconds, when session 2 started, it received almost all of the extra throughput, until the moving average of its recent throughput (shown in Figure 4(b)) caught up to the other stream. A short, quickly dampened period of oscillation occurred. The scheduler ensured that session 1 continued to receive at least its reserved throughput during the startup transient. Similar effects happened at 20 seconds, when session 3 started, and at 30 seconds, when session 2 stopped. Between changes, however, the share of throughput received by each session remained stable.

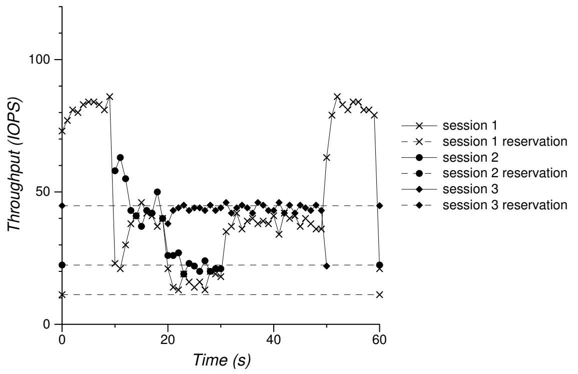
5.3. Overhead

The performance management benefits of Zygaria come at the price of having to perform the scheduling computations, and potentially restricting the set of I/Os that the underlying device has for head scheduling. Each run of the scheduler loop is an $O(n)$ computation in the number of sessions and pools.

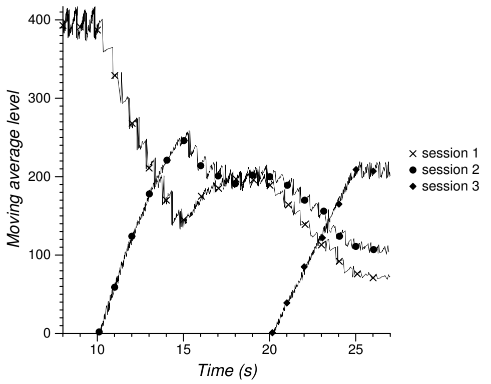
We conducted two experiments to see how the CPU utilization and disk throughput with Zygaria compared to those with the normal Linux disk driver. The first experiment looked at CPU utilization and throughput for random I/Os; the second looked at throughput for concurrent sequential I/O streams.

In the first experiment, we ran 100 I/O generator processes with and without Zygaria. Each generator process performed random I/Os to the whole disk as fast as possible. We compared three basic configurations: without Zygaria; with all hundred generators running against a single session in a single pool; and one generator in each of a hundred sessions, divided as ten sessions in each of ten pools. We compared the first two configurations to determine the basic overhead of Zygaria. The configurations with different numbers of sessions gave an indication of how the performance would scale as the number of sessions increased—recall that the complexity of the scheduler is linear in the number of sessions and pools.

To measure the CPU utilization overhead for each run, we sampled the kernel-reported idle and wait time using the `vmstat` command once a second for 60 seconds, and computed the average amount of non-idle time over those 60 samples. As a result, the accuracy of our CPU overhead measurements was limited by the accuracy of the `vmstat`



(a) Throughput trace



(b) Moving average during first two transitions

Figure 4. Trace of one run, showing throughput sharing over time. Throughput is measured as I/Os per one-second interval; the moving average is sampled on every I/O.

command, and may include effects from our generator processes as well as from Zygaria.

In general, Zygaria introduced a small but measurable CPU overhead, but only a slight difference in throughput, as shown by the following results. The results are the average of 20 runs; 95% confidence intervals are in [brackets].

Configuration	Total CPU usage (%)	Throughput (IOPS)
Without Zygaria	1.17 [0.020]	86.8 [0.17]
1 pool × 1 sess	1.44 [0.023]	86.6 [0.14]
10 pool × 10 sess	1.54 [0.032]	86.9 [0.16]

In the second experiment, we evaluated a system with multiple concurrent sequential I/O streams. Disk head scheduling can be poor in this situation: at worst, the disk head seeks for every I/O request. At best, the head can process many requests together before seeking to where another stream is accessing.

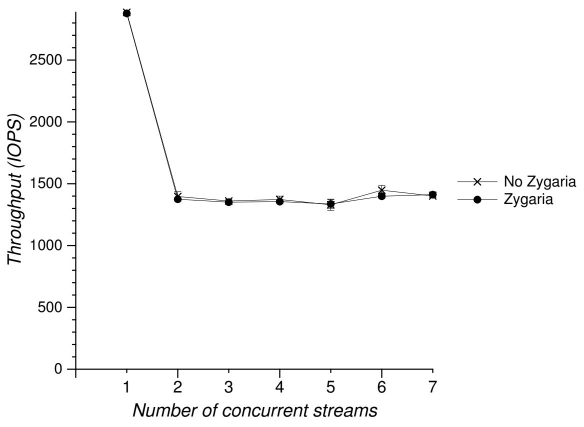


Figure 5. Total throughput across several concurrent sequential streams. Results are the average to 10 runs; error bars show the 95% confidence interval.

We ran between one and seven concurrent sequential workloads for 60 seconds. Each workload was driven by a generator with a closed arrival process that had 20 outstanding 16KB I/O requests and zero think time (*i.e.*, it worked as fast as possible). The *i*th stream of *n* streams started at address 50GB $\cdot\, i/n$, thus spreading the streams widely over the disk surface. When run with Zygaria, each stream had its own session with zero reserved throughput.

Figure 5 shows the total throughput with and without Zygaria. We found that streams received throughput from Zygaria similar to that from the normal Linux disk driver. The experiment did not involve a cache, so pre-fetch and write-behind did not occur.

In general, these results show that the cost of using Zygaria is negligible or low. Moreover, they show that Zygaria is able to manage both random and sequential I/O streams well.

5.4. Mixed workload

The performance management that Zygaria provides has its greatest value for mixed workloads—for example, when some sessions have variable offered load, and others want steady throughput. Our I/O scheduler supports this situation in two ways that simple head schedulers do not: it supports throughput reserves and it smooths out variations in demand by enforcing limits and fair sharing.

To evaluate how well Zygaria handles mixed workloads, we constructed a synthetic workload that generated a mixture of media-like, transaction-like and housekeeping-like I/O streams. Some streams were sequential, some random; some presented constant demand, others presented variable demand. We constructed the streams with the same workload generator used in our other experiments.

The mixed workload consisted of the following three types of streams:

- **background:** modeled random-access management traffic, such as a file system backup. One pool, with reserve 10% and limit 25%.
- **media:** modeled three constant bit rate media accesses with three constant-rate sequential streams in one pool. One pool with reserve 30% and limit 35%; three equal sessions in that pool.
- **transaction:** modeled short bursts of random-access traffic, such as a transaction-processing application might generate. Two pools, each with a 30% reserve and no limit. Within a pool, bursts arrived at a random time and lasted for a random duration. Each burst had its own session. The burst arrivals were admission-controlled to ensure that sessions remained feasible.

The total offered load was almost enough to saturate the disk. A script drove the generators, so that the sequence of transaction burst arrivals could be executed repeatedly.

Figure 6 shows traces of the resulting throughput with and without Zygaria. In general Zygaria ensured that the streams did not interfere with each other.

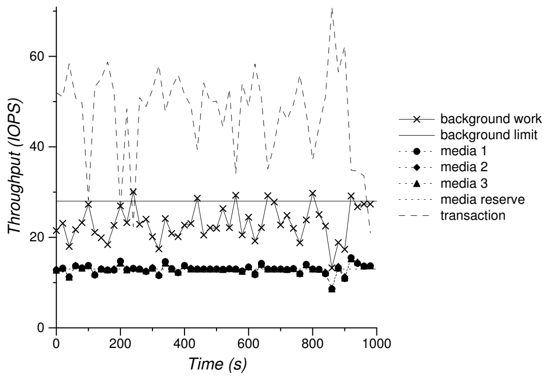
The transaction-processing workloads exhibited variable demand. Unregulated, these swamped the system and interfered with other streams—often, there would be more I/O requests outstanding from the transaction streams than from the other streams combined. With Zygaria, the transaction streams completed the same amount of work but the rates were throttled to ensure the other streams received their share of throughput.

The media streams, which had a narrow range of acceptable throughput between their reserve and limit, received their proper throughput with Zygaria, regardless of other activity in the system.

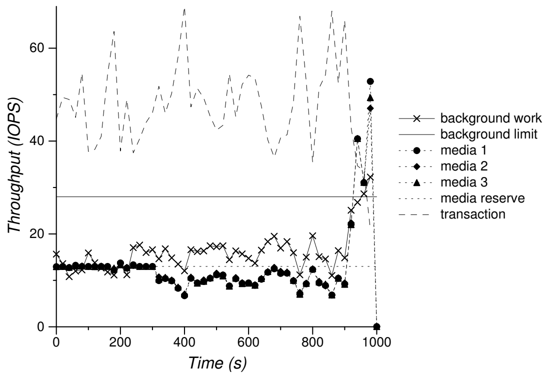
The background stream was designed to receive a modest minimum throughput, but soak up a share of best-effort throughput. It completed more work with Zygaria than without, partly because Zygaria ensured that it received its reserve, but mostly because the fair sharing algorithm allocated its pool a share of throughput equal to each of the two transaction-processing pools.

The greatly improved responsiveness of the media applications demonstrated that one can use Zygaria to achieve an effect like traditional priority-based schedulers: by giving the media streams a reserve that matched their I/O demands, Zygaria would schedule their I/Os immediately as the application submitted them, ahead of I/Os from the other applications.

Overall, this experiment showed that Zygaria handles mixed workloads well.



(a) With Zygaria



(b) Without Zygaria

Figure 6. Trace of one 1000-second run, showing the behavior of multiple workloads over a longer term, with and without Zygaria. Points shown are the average for 20-second intervals.

6. Related work

The problem of managing I/O performance resources can be divided into two separable problems: how to specify allocations for pools and sessions, and then how to deliver on those allocations. Delivering performance resources combines issues of soft real-time scheduling for fulfillment of reserves and of sharing extra resources fairly. For all these aspects, our I/O scheduling algorithm builds on the large body of previous work.

Resource allocation. Our algorithm uses a hierarchical pool and session model with reserves and limits to manage resource allocation. These notions of hierarchical structure and of reserve/limit specifications have precedent in QoS allocation models.

Traditional QoS resource allocation models support potentially multiple levels of specification—for example, a reserve, a limit, and points in between. For each level, the specification sets the performance that the system must guarantee. Simple models support only a single level, and use metrics such as bandwidth (first seen in XFS [8]) to express requirements. More complex models use benefit-value (DQM [3]) or utility functions (Q-RAM [19]) to express requirements, and the system uses these functions to maximize the overall benefit or utility over all applications while ensuring that minimum levels are met; the user or application must specify the function, which is often difficult. We opt for a simple QoS specification: a minimum QoS level for reserve and a maximum QoS level for limit.

Several hierarchical allocation models exist for resource management. Generalized models exist for the management of multiple resources, including Eclipse [5] and hierarchical Q-RAM. Models also exist for CPU scheduling [11, 20] and network sharing [1, 9]. Most of these examples support arbitrary hierarchy depths; for our purposes, a two-level hierarchy suffices.

Our allocation model is most similar to the one used by Wu *et al.* [29]. Their I/O scheduling algorithm uses an arbitrary hierarchy of token buckets to provide proportional resource guarantees to applications. It allows applications to borrow performance from other applications that are not using their share of performance, but does not address fair sharing of best-effort performance. Their algorithm requires *a priori* knowledge of the actual device throughput under the current workload, whereas ours only requires the worst-case throughput.

Soft real-time scheduling. Our algorithm supports soft real-time scheduling. As such, we guarantee that the average throughput will meet the reserves specified for pools and sessions over the long term, but allow occasional violations in the short term. Our work draws upon soft real-time scheduling algorithms for various kinds of resources.

Several projects have investigated disk schedulers that support a mix of multimedia and non-multimedia applications. Clockwise [2] handles a combination of both periodic real-time and best-effort streams. It gives priority to best-effort streams, delaying real-time I/Os as long as possible without violating their requirements. Cello [22], MARS [6], and the work by Wijayaratne and Reddy [27] all implement a two-level hierarchy of schedulers for multiple classes of traffic. Compared to these systems, we only guarantee the fulfillment of reserves on average. On the other hand, we do not require detailed information (such as their periodicities) about the application workloads.

Other work on disk scheduling for multimedia applications often assumes that no other applications will access the storage, which allows for greater optimization in algorithm design. A survey paper by Gemmell *et al.* covers several representative systems [10].

Response time control. In our work, we focus on reserve and limit throughput specifications. Related systems exist to control other storage system characteristics, most notably response time. Façade [16] uses an EDF scheduler that bases the deadline of an I/O on the response time requirement its stream, with adaptive mechanisms to adjust the response time target as the offered load of the stream changes. SLEDS [7] provides per-stream I/O rate throttling so that all streams will receive specified response latencies. SLEDS is adaptive: a central server monitors the performance each stream is receiving and changes the acceptable rates for other streams when one stream is getting response time longer than its requirement.

Fair and proportional sharing. Several alternatives exist for sharing performance resources from storage devices, many of which are related to methods for sharing CPU cycles and network bandwidth. The most obvious is lottery scheduling [24, 26], which supports proportional sharing of resources among multiple users, and includes a hierarchical approach for defining the shares. Another is Start-time Fair Queuing (SFQ) [12] and its successors YFQ [4], SFQ(D) [13], and FSFQ(D) [13], which give each active I/O stream a share of resources in proportion to its weight relative to any other active streams. Our algorithm differs from these alternatives in two ways: it gives each active stream its requested reserve of resources regardless of the demands of other streams, and (in contrast to lottery scheduling) it is deterministic in its scheduling decisions.

Underlying devices. We have assumed that the underlying device has an approximately constant average seek time. This behavior is in contrast to more complex devices such as Iceberg [23] or AutoRAID [28], whose seek time may change as the data layout changes. For such devices, we expect that the performance benefits of our scheduling algorithm would be gained by incorporating it into the device.

7. Conclusions

We have implemented an I/O scheduler that provides isolation between the pools of applications that share resources on a storage device, and also between the sessions of clients of the same application. It accomplishes these goals through the enforcement of simple reserve and limit policies on performance resource allocation. The scheduler guarantees a reserve I/O rate to each session and pool, limits each session and pool to a maximum I/O rate, and provides fair sharing of any available unused resources. It is implemented in a Linux kernel module over real underlying disks, and causes little overhead. We have verified that the implementation operates as we had intended by experimentation.

Our implementation can be used as one of the building blocks to construct a large, scalable storage system that is built from small storage bricks. Such a system can in turn be used to aggregate the data and workloads of multiple applications onto a cluster of storage systems. Our experimental results give us confidence that brick-level enforcement of resource allocation policies can translate into cluster-wide isolation of applications.

Acknowledgments

We thank Omer Zaki, who implemented an early version of this algorithm, and Scott Brandt and other members of the Storage Systems Research Center, UC Santa Cruz, for helpful discussions that refined our algorithm. We also thank the anonymous reviewers for their feedback.

References

[1] J. C. R. Bennett and H. Zhang. Hierarchical packet fair queuing algorithms. *IEEE Trans. on Networking*, 5(5):675–689, Oct. 1997.

[2] P. Bosch and S. J. Mullender. Real-time disk scheduling in a mixed-media file system. In *Proc. of the IEEE Real-Time Technology and Applications Symp.*, pages 23–32, June 2000.

[3] S. Brandt and G. Nutt. Flexible soft real-time processing in middleware. *Real-Time Systems*, 22(1–2):77–118, Jan. 2002.

[4] J. L. Bruno, J. C. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *Proc. of the 1999 IEEE Intl. Conf. on Multimedia Computing and Systems*, pages 400–405, July 1999.

[5] J. L. Bruno, E. Gabber, B. Özden, and A. Silberschatz. The Eclipse operating system: Providing quality of service via reservation domains. In *Proc. of the USENIX Annual Technical Conf.*, pages 235–246, June 1998.

[6] 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.

[7] 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, Oct. 2003.

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

[9] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE Trans. on Networking*, 3(4):365–386, Aug. 1995.

[10] D. J. Gemmell, H. M. Vin, D. D. Kandlur, P. V. Rangan, and L. A. Rowe. Multimedia storage servers: A tutorial. *IEEE Computer*, 28(5):40–49, May 1995.

[11] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*, pages 107–121, Oct. 1996.

[12] 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.

[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, June 2004.

[14] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proc. of the 13th IEEE Real-Time Systems Symp.*, pages 110–123, Dec. 1992.

[15] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[16] C. R. Lumb, A. Merchant, and G. A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proc. of the 2nd Conf. on File and Storage Technology*, pages 131–144, Mar.–Apr. 2003.

[17] C. R. Lumb, J. Schindler, G. R. Ganger, D. F. Nagle, and E. Riedel. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation*, pages 87–102, Oct. 2000.

[18] A. V. Oppenheim and A. S. Willsky. *Signals and Systems*. Prentice-Hall, 1983.

[19] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. A resource allocation model for QoS management. In *Proc. of the 18th IEEE Real-Time Systems Symp.*, pages 298–307, Dec. 1997.

[20] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proc. of the 22nd IEEE Real-Time Systems Symp.*, pages 3–14, Dec. 2001.

[21] S. Shenker, C. Partridge, and R. Guerin. Specification of guaranteed quality of service. RFC 2212, IETF, 1997.

[22] P. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proc. of SIGMETRICS 1998, the Intl. Conf. on Measurement and Modeling of Computing Systems*, pages 44–55, June 1998.

[23] Storage Technology Corporation. Iceberg 9200 Storage System: Introduction. STK Part Number 307406101, Storage Technology Corporation, 1994.

[24] D. G. Sullivan and M. I. Seltzer. Isolation with flexibility: A resource management framework for central servers. In *Proc. of the USENIX Annual Technical Conf.*, pages 337–350, June 2000.

[25] T.-S. Tia, J. W. Liu, and M. Shankar. Algorithms and optimality of scheduling aperiodic requests in fixed-priority preemptive systems. *Real-Time Systems*, 10(1):23–43, Jan. 1996.

[26] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation*, pages 1–11, Nov. 1994.

[27] 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.

[28] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *Trans. on Computer Science*, 14(1):108–136, Feb. 1996.

[29] J. Wu, S. Banachowski, and S. A. Brandt. Hierarchical disk sharing for multimedia systems. In *Proc. of the 15th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 189–194, June 2005.