Sweet Storage SLOs with Frosting

Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Ion Stoica, Randy Katz
University of California, Berkeley

Abstract

Modern datacenters support a large number of applications with diverse performance requirements. These performance requirements are expressed at the application layer as high-level *service-level objectives* (SLOs). However, large-scale distributed storage systems are unaware of these high-level SLOs. This lack of awareness results in poor performance when workloads from multiple applications are consolidated onto the same storage cluster to increase utilization. In this paper, we argue that because SLOs are expressed at a high level, a high-level control mechanism is required. This is in contrast to existing approaches, which use block- or disk-level mechanisms. These require manual translation of high-level requirements into low-level parameters. We present Frosting, a request scheduling layer on top of a distributed storage system that allows application programmers to specify their high-level SLOs directly. Frosting improves over the state-of-the-art by automatically translating high-level SLOs into internal scheduling parameters and uses feedback control to adapt these parameters to changes in the workload. Our preliminary results demonstrate that our overlay approach can multiplex both latency-sensitive and batch applications to increase utilization, while still maintaining a 100ms 99th percentile latency SLO for latency-sensitive clients.

1 Introduction

Modern datacenter architectures must support a variety of both user-facing and internal applications. Each application typically has an associated performance requirement, expressed in terms of a *service-level objective* (SLO), such as a latency or throughput target for end-to-end requests. SLOs reflect the expectations of the users of that application and violations can result in significant economic repercussions [18].

However, today's distributed storage systems, such as HBase and BigTable [1, 6], are not directly aware of applications' high-level SLOs. Current solutions require manual tuning of low-level internal system parameters like weights, slots, or disk IOPS until the desired high-level SLO is satisfied [10]. Manually setting static parameters is painful for application programmers, who are forced to translate their high-level SLOs into these foreign low-level system parameters. Moreover, static parameters fundamentally fail to adapt to the rapid and frequent workload changes in the datacenter. Critically, manual tuning inevitably becomes unmanageable as system complexity and the number of workloads increase.

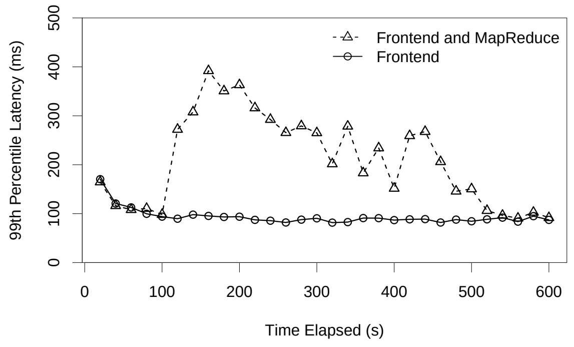

Figure 1: A latency-sensitive frontend application issues `get` requests to a static HBase configuration. A MapReduce job begins issuing `scan` requests to the same HBase cluster at time $t = 100$ and finishes at $t = 500$. Frontend latency increases by 3x due to contention for the storage layer.

These shortcomings stymie the ability of datacenters to multiplex different workloads onto a single, shared storage layer. Because of this inability to multiplex, datacenters often contain multiple, physically distinct storage systems, each provisioned separately. This has significant economic costs. First, each separate storage system must be provisioned individually for peak load. This requires a higher degree of *overprovisioning*, and contributes to *underutilization* of the cluster [2]. Second, segregation of data leads to *data staleness*. Analytics jobs typically must wait for the freshest data to be copied into a batch system. This lag results in delayed insights and suboptimal results [3]. Copying also leads to duplication of data and the associated consistency and management concerns. Finally, there is increased *operational complexity*, requiring additional staff and expertise, and increasing exposure to software bugs and configuration errors [13].

Consolidating multiple workloads onto a single storage system avoids these problems. However, a naïve approach to consolidation does not realize these benefits, since unmediated resource demands lead to SLO violations. In an initial experiment, we ran a latency-sensitive frontend application and a MapReduce job, both simultaneously issuing requests against the same HBase cluster. The results of this experiment (Figure 1) show that the latency-sensitive frontend application can suffer a 3x increase in 99th percentile latency when run alongside a batch job. Companies like Amazon, Google, and Microsoft have identified that degradation in the 99th percentile latency is a major source of user dissatisfaction [12]. Thus, to control the end-to-end 99th percentile latency while consolidating workloads, we focus on high-level SLOs, which marks a shift from existing solutions based on low-level operations.

However, there are a number of challenges to overcome to achieve this goal.

Translating high-level SLOs. The storage system should allow programmers to specify their high-level SLOs directly, and automatically translate these SLOs into the correct set of internal system parameters.

Scheduling. The lack of preemption and skew in request size differentiates storage request scheduling from other domains. This is exacerbated by how distributed storage systems sit atop a deep software and hardware stack (Figure 2). Consolidating storage workloads requires choosing appropriate scheduling mechanisms at the right layers in the stack.

Dynamic SLO enforcement. Due to diurnal patterns and load fluctuations, the storage system needs to adjust request scheduling in response to the workload. This dynamism will allow the system to take advantage of slack in the workload to run low-priority requests, while also being able to gracefully adapt to load spikes.

To address these concerns, we present Frosting. Frosting adopts a new top-down approach to enforcing SLOs. Frosting acts as an overlay atop an existing distributed storage system, performing scheduling on requests from different applications to enforce SLOs. An application's high-level SLOs are translated into proportional share allocations by a feedback controller. These allocations are used to weight the admission of a client's requests to the underlying storage system. The feedback controller adapts to dynamic changes in the workload by monitoring each client's performance, and continually adjusting allocations to maximize overall SLO compliance of the system. Our preliminary results demonstrate that our overlay approach can multiplex both latency-sensitive and batch applications to increase utilization, while still maintaining a 100ms 99th percentile latency SLO for latency-sensitive clients. Finally, we discuss further improvements and potential avenues of future work.

2 Design

Frosting is designed to overlay distributed storage systems which support `get`, `put`, and `scan` operations on rows or objects. An example architecture is shown in Figure 2. Frosting is backwards-compatible with the API of the underlying storage system. Clients with an associated SLO tag their requests to Frosting with their client name. Legacy untagged requests are treated as low-priority, and are handled with a best-effort FIFO policy. High-level SLOs are defined as a percentile latency or throughput target for a given type of request from a client name. For example, an application could tag all of its requests with

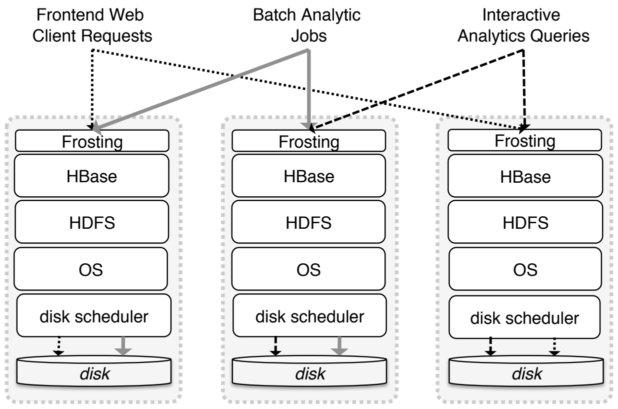

Figure 2: Frosting running on a 3-node cluster. By scheduling `get`, `put`, and `scan` requests, controlling when and what order they are released to HBase, Frosting limits queuing in lower layers of the software and hardware stack.

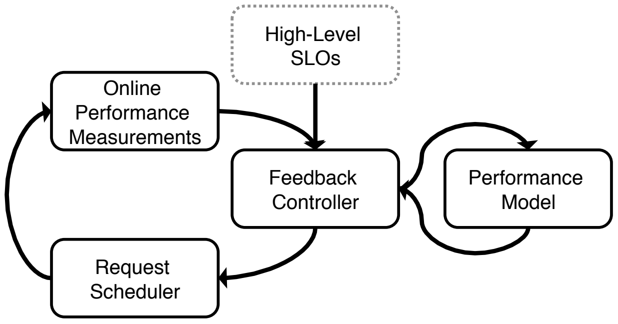

Figure 3: Frosting design: High-level SLOs are translated by the feedback controller (§2.3) into parameter settings in the request scheduler (§2.2).

the client name `twitter`, and specify a 99th percentile latency SLO of 200ms for `puts` and a throughput SLO of 10 qps for `scans`.

These high-level SLOs are mapped into actual scheduling decisions by the feedback controller, shown in Figure 3. Frosting monitors each client's SLO compliance, and use online performance measurements to build a linear performance model that predicts how a client's performance will change with changes in scheduler allocation. These performance models are used by the feedback controller to continually adjust each client's allocation to maximize SLO compliance of the system. Frosting's scheduling mechanism and dynamic control scheme are detailed in §2.2 and §2.3.

2.1 System Model

Frosting ensures predictable performance by limiting the number of outstanding requests in the underlying software and hardware stack. Limiting the number of outstanding requests minimizes the degree of queuing in lower lay-

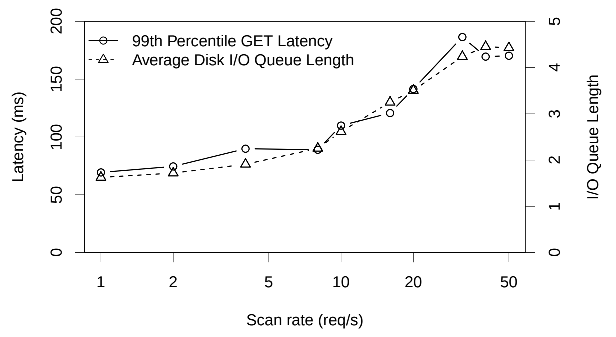

Figure 4: A fixed number of `gets` were issued to the HBase cluster, while the number of `scans` were slowly increased. At a rate of 8 or fewer `scans` per second, queue lengths in lower layers are short, keeping latency low.

ers, which is important since queuing delay tends to dominate total latency in this type of system [11]. In fact, as Frosting has no way of preempting or controlling a request once it has been issued to a lower layer, keeping the queue lengths in lower layers short is the only way to bound latency. However, limiting queue length means the system operates below peak utilization.

Figure 4 demonstrates the effect of queueing within the stack. We performed an experiment with two clients, the first issuing a constant rate of latency-sensitive reads, and the second issuing an increasing rate of large scans. As the rate of scans increases above 8 req/s, the length of the block I/O queue in the operating system (as measured by `iostat`) increases rapidly, along with 99th percentile read latency. To preserve latency, Frosting attempts to keep queue lengths below this operating point.

The overlay scheduling approach we chose with Frosting is simple and minimizes changes to existing code. A coordinated scheduling approach, which performs request scheduling at multiple layers of the stack, may be able to achieve better latency and throughput properties. However, out initial evaluations of Frosting show that even an overlay approach can result in meaningful improvements. We plan to explore the strengths and weaknesses of both approaches as future work.

2.2 Scheduling

With the feedback controller enforcing policy in the form of user-stated SLOs, the scheduler becomes merely a mechanism. The feedback controller adjusts the scheduler's parameters to effect a corresponding change in a client's performance. The Frosting scheduler is thus unconcerned with traditional scheduling goals of maximizing fairness or interactivity. Instead, its primary requirement is to behave predictably to changes made by the feedback controller, with the secondary goal of simplicity of API.

2.2.1 Storage request scheduling

Request scheduling for distributed storage systems is a challenging problem which differs from the classical domains of CPU and packet scheduling in two respects.

Lack of preemption. CPU schedulers use preemption to limit the amount of CPU time spent running a thread. A request scheduler is unable to preempt requests because it is impossible to "unsend" or cancel a request to a lower layer of the stack. This lack of preemption leads to greater degrees of skew in allocation in the system, which complicates allocating exact portions of system resources.

Size skew. In packet scheduling, packets are quantized units up to an MTU in size, and require processing time directly proportional to size. Storage requests to Frosting are not chunked into MTUs, and can take unknown amounts of processing time. A large 1000-row scan is viewed as a single request, resulting in a long blocking operation of unknown length. The actual processing time depends on the type of request, composition and magnitude of the workload, and underlying system properties.

2.2.2 Frosting Scheduler

Scheduling in Frosting can be modeled abstractly as a set of worker threads pulling requests from a shared queue. Incoming requests are pushed into this queue, and popped off by worker threads. Internal to this abstract queue, the Frosting scheduler separates requests into additional queues based on client name and request type. Proportional share scheduling is used to decide whose request to serve [23]. Proportional share supports fine-grained, fractional allocation of resources. It is convenient for situations with a dynamic set of clients, since relative weightings always remain consistent.

The Frosting prototype implements proportional share via lottery scheduling [23]. Each time a worker thread becomes inactive, a lottery is held among all queues with pending requests. The lottery is biased based on the proportional share assigned to each queue, such that a queue with twice the share will have twice as many of its requests dequeued.

This ignores the actual cost of handling a request, and is most predictable when all queues have outstanding requests. However, as shown in §3, even this simple scheme shows benefits under the assumption of closed-loop clients. We discuss other potential scheduler improvements in §5.

2.3 Dynamic Control

The feedback controller in Frosting is used to adapt to changes in the workload. The controller actively monitors each client's normalized SLO performance, which is expressed as the ratio between a client's desired and actual performance. For example, a client with an SLO of 100ms experiencing 200ms of actual latency has a normalized SLO performance of 0.5. The feedback controller periodically adjusts each client's proportional share allocation to maximize overall SLO compliance of the system. These changes are made based on an online linear performance model of each client built from performance measurements collected in prior time intervals. Since these models are recomputed each time interval, they approximate the actual non-linear behavior of the system.

Solving for client allocations $a_i, ..., a_n$ for an approaching time interval t can be expressed as a linear program, originally formulated by Merchant, et al. [15]:

$$\forall_{i,j,i \neq j} w_i(1 - p_i(a_i^t)) - w_j(1 - p_j(a_j^t)) < \epsilon$$

The $(1 - p_i(a_i^t))$ term expresses a "badness" factor for client i in terms of normalized SLO performance, with potential allocation a_i^t and the client's linear performance model p_i. This badness factor is then weighted by a w_i term that indicates the relative importance of the client.

The linear program attempts to minimize ϵ, the maximum difference in badness of all pairs of clients i, j. During overload, this has the effect of proportionally degrading each client's performance according to their weight.

Two additional constraints are necessary. Changes in allocation are limited to a step size σ, which prevents the feedback controller from solving for allocation points where the linear performance models do not apply. Total allocation by the system must also sum to 1.

$$\forall_i |a_i^t - a_i^{t-1}| \leq \sigma$$
$$a_1^t + ... + a_n^t = 1$$

3 Evaluation

The prototype implementation of Frosting is built on top of HBase, a distributed column-store similar to Google's BigTable [1, 6]. Modifications to existing code were minimal. It was straightforward to replace the existing FIFO request queue with our own abstract queue with overloaded enqueue and dequeue methods. The HBase RPC protocol was also modified to include the client's name with each request. No modifications were necessary to HDFS or the operating system.

We evaluate two components of Frosting. First, we evaluate the suitability of the Frosting proportional share scheduler as a mechanism (§2.2). We test how setting different proportional share values affect a latency-sensitive client's 99th percentile SLO when Frosting is under contention from a batch workload. Next, we use the same situation to evaluate the ability of the feedback controller to enforce policy by setting a high-level SLO for the latency-sensitive client (§2.3). All experiments were run on a 3-node cluster of EC2 c1.xlarge instances, with HDFS configured to use the 4 local disks on each instance. A Yahoo! Cloud Serving Benchmark [7] (YCSB) client was used to issue latency-sensitive read requests to Frosting with a uniformly random key access pattern. A MapReduce word count job with 4 map tasks was used to generate batch scan requests to Frosting over the same key range as the YCSB client.

Proportional share scheduler. To provide a baseline, we measured YCSB latency with HBase's default single FIFO queue without MapReduce running concurrently. We then ran three experiments with concurrent MapReduce: the Frosting scheduler with share settings of 90:10 and 99:1 for YCSB to MapReduce, as well as HBase's default FIFO policy again. Each experiment was run 10 times. Figure 5(a) plots the mean and standard deviation in 99th percentile latency for each run. Figure 5(b) shows the average MapReduce throughput for each.

We see that the Frosting scheduler performs superiorly to FIFO when YCSB is under contention with MapReduce. Both proportional share settings show little fluctuation over the duration of the experiment, and are able to achieve better latencies than FIFO. While FIFO increases to as much as twice the baseline 99th percentile latency, the 99:1 policy remains within 10% of baseline while increasing utilization by allowing some MapReduce requests to run. 90:10 strikes a point in between FIFO and 99:1, allowing more MapReduce requests to run at the cost of increased latency. This demonstrates that proportional share allocations predictably map to 99th percentile latency values, and shows how Frosting can tradeoff between two competing clients.

Feedback control. Given that proportional share is an appropriate mechanism, we evaluate the ability of the feedback controller to converge on and maintain a high-level SLO. We specified a 99th percentile read SLO of 100ms for YCSB to Frosting, with a weight of $w_1 = 80$. MapReduce was given a throughput SLO of 40 req/s with a lower weight of $w_2 = 1$. This throughput SLO allows MapReduce to balloon to use any extra capacity in the system as long as YCSB's SLO is being met.

Figure 5(c) depicts how the feedback controller adjusts YCSB's proportional share, and YCSB's 99th percentile latency. YCSB and MapReduce initially start off with equal share allocations. The feedback scheduler increases YCSB's allocation at the maximum step size (σ), until YCSB meets its 99th percentile latency SLO around $t = 70$. Since YCSB briefly exceeds its performance target here, share oscillates as the controller trades off between YCSB and MapReduce, before settling around a value of 90. The inherent noisiness in 99th percentile latency results in occasional variation in YCSB share and latency, even if share allocations do not change. We plan to explore further modifications to the feedback controller to mitigate this effect.

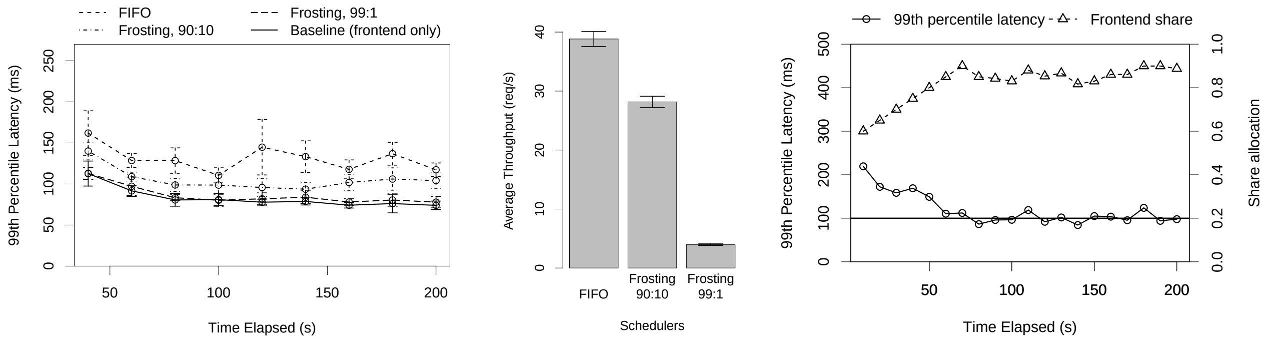

Figure 5: In two experiments, a latency-sensitive frontend workload (YCSB) and a throughput-oriented batch workload (MapReduce) issue requests to the same HBase cluster. *Left and middle:* Here, the Frosting share parameters are set to a fixed known value. We compare the latency (left) and throughput (right) to those obtained by the baseline and FIFO. Performance of the two clients varies predictably as the share values are adjusted. *Right:* When a 100ms YCSB SLO is specified, the feedback controller rapidly increases YCSB's share until the SLO is met at $t = 70$. The controller converges to approximately a 90:10 setting.

4 Related Work

Many others have looked at consolidating applications onto shared storage resources. Soundararajan et al. present an approach to controlling storage bandwidth contention in a server farm by passing application-level quality-of-service requirements across to a SAN and down to the OS [20]. Their approach requires modification to the kernel.

Padala et al. also focus on controlling resource contention among applications, but use a hypervisor-level approach involving controlling CPU allocations to storage clients [17]. Similarly, Soundararajan et al. present a technique for coordinated partitioning of storage and memory bandwidth in a shared, virtual storage environment to prioritize among latency-sensitive applications [21]. We differ in that we support latency SLOs while also scheduling throughput-oriented requests to improve utilization.

Argon provides performance isolation for shared storage clients also using cache partitioning, request amortization, and quanta-based disk time scheduling, but focuses on providing guarantees for throughput rather than high-percentile latency [22].

PARDA, mClock, and Maestro provide SLOs for virtual machines and storage arrays [9, 10, 15]. Differentiated storage attaches semantic information to storage requests to associate specialized caching policies [16]. These systems differ from ours by operating at the block device level and not incorporating high-level SLOs.

Many other systems have also focused on scheduling disk I/O for differentiated quality-of-service [4, 5, 8, 14, 19]. They differ from Frosting in that they provide guarantees on operations to the block device, rather than requests to an application-level storage system.

5 Future Work

We are considering multiple avenues of future work.

Additional scheduler parameters. Proportional share scheduling can be combined with allocation *reservations* and *limits* to express stronger policies for certain workload scenarios. Another potential parameter is adjusting the total number of worker threads in the system. However, these additional parameters are more difficult to model dynamically and fit within a linear program.

Coordinated stack scheduling. Controlling request scheduling at each layer of the software stack, while more complicated, should perform better than a pure overlay scheduler like Frosting. A coordinated approach can also better manage "multicast" effects, where a single request to a layer spawns multiple requests to lower layers.

Storage system interfaces. Consolidating onto a shared storage layer requires clients to use a common storage API. We chose to use HBase in Frosting, but a client might wish to also consolidate MapReduce running directly on HDFS, or MySQL directly on the block device. Settling on a common storage abstraction is necessary to support a broad range of clients.

Economic argument. There is an inherent tradeoff between throughput and latency in any queuing system. An open problem is understanding when consolidation reduces monetary provisioning costs and quantifying this economically.

Acknowledgements

This research is supported in part by an NSF CISE Expeditions award, gifts from Google, SAP, Amazon Web Services, Blue Goji, Cisco, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, Intel, MarkLogic, Microsoft, NetApp, Oracle, Quanta, Splunk, VMware and by DARPA (contract #FA8650-11-C-7136).

References

[1] Hbase. `http://hbase.apache.org`.

[2] L. A. Barroso. Warehouse-Scale Computing: Entering the Teenage Decade. In *ISCA '11*.

[3] D. Borthakur et al. Apache hadoop goes realtime at facebook. In *SIGMOD '11*.

[4] J. Bruno et al. Disk scheduling with quality of service guarantees. In *ICMCS '99*.

[5] D. Chambliss et al. Performance virtualization for large-scale storage systems. In *SRDS '03*.

[6] F. Chang et al. Bigtable: A distributed storage system for structured data. *ACM TOCS*, 2008.

[7] B. Cooper et al. Benchmarking cloud serving systems with YCSB. In *SOCC '10*.

[8] P. Goyal, H. Vin, and H. Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *Networking*.

[9] A. Gulati, I. Ahmad, and C. Waldspurger. PARDA: Proportional allocation of resources for distributed storage access. In *FAST '09*.

[10] A. Gulati, A. Merchant, and P. Varman. mclock: Handling throughput variability for hypervisor io scheduling. In *OSDI '10*.

[11] A. Gulati et al. BASIL: Automated IO load balancing across storage devices. In *FAST '10*.

[12] J. Hamilton. The cost of latency. `http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx`.

[13] Y. Izrailevsky. Nosql at netflix. `http://techblog.netflix.com/2011/01/nosql-at-netflix.html`.

[14] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: Virtual storage devices with performance guarantees. In *FAST '03*.

[15] A. Merchant et al. Maestro: quality-of-service in large disk arrays. In *ICAC '11*.

[16] M. P. Mesnier and J. B. Akers. Differentiated storage services. *SIGOPS Operational System Review*.

[17] P. Padala et al. Adaptive control of virtualized resources in utility computing environments. In *EuroSys '07*.

[18] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search, 2009.

[19] P. J. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. In *SIGMETRICS '97*.

[20] G. Soundararajan and C. Amza. Towards end-to-end quality of service: Controlling i/o interference in shared storage servers. In *Middleware '08*.

[21] G. Soundararajan et al. Dynamic resource allocation for database servers running on virtual storage. In *FAST '09*.

[22] M. Wachs et al. Argon: performance insulation for shared storage servers. In *FAST '07*.

[23] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *OSDI '94*.