Performance Differentiation for Multi-port Arrays: A Control-Theoretic Approach

Pradeep Padala
University of Michigan
ppadala@eecs.umich.edu

Mustafa Uysal
HP Labs
mustafa.uysal@hp.com

Arif Merchant
HP Labs
arif.merchant@hp.com

Xiaoyun Zhu
VMware
xzhu@vmware.com

Sharad Singhal
HP Labs
sharard.singhal@hp.com

Kang Shin
University of Michigan
kgshin@eecs.umich.edu

Abstract

Large multi-port disk arrays typically store data for multiple applications with diverse performance requirements. Each application has a different priority, representing the relative importance of the application to the business. Currently, performance differentiation and isolation for storage workloads is implemented by statically partitioning the disk array resources, which results in over-provisioning and under-utilization of resources. In this paper, we present a feedback controller for storage workloads to provide performance differentiation among multiple applications. Our feedback controller monitors the performance of each application and dynamically allocates the array resources so that diverse performance requirements can be met without static partitioning. The controller consists of three layers, a set of application controllers, an arbiter and a port allocator. The application controller uses an online auto-regressive model to determine the I/O resources required to meet an application's target. The arbiter employs linear programming to arbitrate among multiple applications based on priorities, in the case of an overload. The port allocator uses each application's demand through a particular port to determine per-port scheduler settings. Our preliminary experiments indicate that the controller can dynamically adjust the per-port scheduler parameters to achieve application targets. The controller can also enforce application priorities and can handle both throughput and latency metrics.

1. INTRODUCTION

The common mode of running applications in corporate environments is to consolidate them onto shared hardware in large data centers. This allows benefits such as improved utilization of the resources, higher performance, and centralized management. However, as more applications compete for shared resources, the system has to be flexible enough to enable each application to meet its performance requirements.

Applications running on the shared storage present very different storage loads and have different performance requirements: for example, Online Transaction Processing (OLTP) applications might present bursty loads and require bounded IO response time; business analytics may require high throughput; and back-up applications usually present intense, highly sequential workloads with high throughput requirements. When the requirements of all applications cannot be met, the choice of which application requirements to meet and which ones to abandon may depend upon the priority of the individual applications. For example, meeting the I/O response time requirement of an interactive system may take precedence over the throughput requirement of a backup system. A data center operator needs the flexibility to set the performance metrics and priority levels for the applications, and to adjust them as necessary.

The proposed solutions to this problem in the literature include using proportional share I/O schedulers (e.g., SFQ [7]) and admission control (I/O throttling) using feedback controllers (e.g., Triage [9]). Proportional share schedulers divide the throughput available from the storage device between the applications in proportion to the applications' shares (a.k.a. weights), which are set statically by an administrator. Such schedulers alone cannot provide application performance differentiation, for several reasons: 1) the application performance depends on the proportional share settings and the workload characteristics in a complex, non-linear, time-dependent manner, and it is difficult for an administrator to determine in advance the share to assign to each application, and how/when to change it; 2) applications have several kinds of performance targets, such as response time and bandwidth requirements, not just throughput; and 3) in overload situations, when the system is unable to meet all of the application QoS requirements, prioritization is necessary to enable important applications to meet their performance requirements.

In this paper, we present an approach that combines an optimization-based feedback controller with an underlying IO scheduler. Our controller accepts performance metrics and targets from multiple applications, monitors the performance of each application, and periodically adjusts the IO resources given to the applications at the disk array to make sure that each application meets its performance goal. The performance metrics for the applications can be different: the controller normalizes the application metrics so that the

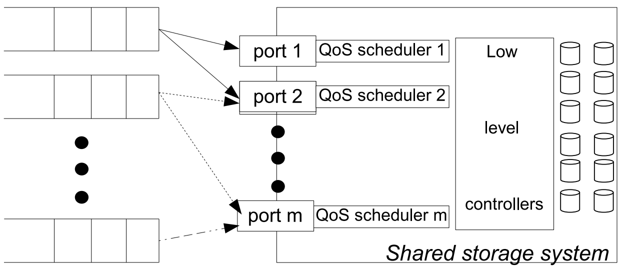


Figure 1: Storage system consists of a shared disk array with independent proportional share I/O scheduler running at each port.

performance received by different applications can be compared and traded off. Our controller continually models the performance of each application relative to the resources it receives, and uses this model to determine the appropriate resource allocation for the application. If the resources available are inadequate to provide all the applications with their desired performance, a Linear Programming optimizer is used to compute a resource allocation that will degrade each application's performance in inverse proportion to its priority. Initial evaluations using this controller with a large commercial disk array show very promising results.

2. STORAGE CONTROLLER DESIGN

In this section, we first describe our system model. We then present our design of the storage controller and describe its components in detail.

2.1 System model

Our system consists of a disk-array with a number of input ports where applications submit their I/O requests. The ports all share the same back-end resources and applications can use any combination of ports. Each port has an independent concurrency-limiting I/O scheduler, which controls the sharing among the I/O requests passing through its port. Once scheduled, an I/O request is released to the back-end of the disk array to access a shared array cache and disk devices. Each application using the system belongs to a service class used to indicate the desired level of service. Each service class has its own specified performance target (either I/O throughput or latency) and a priority level. In order to meet the QoS targets of the application classes, an external feedback controller periodically polls the disk array to determine the performance each class is receiving and then adjusts the parameters of all the schedulers running at each port of the disk array to meet the performance targets.

In our current system, the port IO schedulers have a single parameter: a per-application-class *concurrency bound*. The scheduler limits the number of IO requests outstanding at the disk array back-end from each application class to its concurrency bound. For example, if an application class has a concurrency bound of 2, and it has two I/O requests pending at the back-end, the scheduler will not send any more requests from that class to the back-end until at least one of the pending requests finishes. The total concurrency of the array (i.e., the total number of IOs permitted at the back-end from all ports) is limited, either by the system, or by

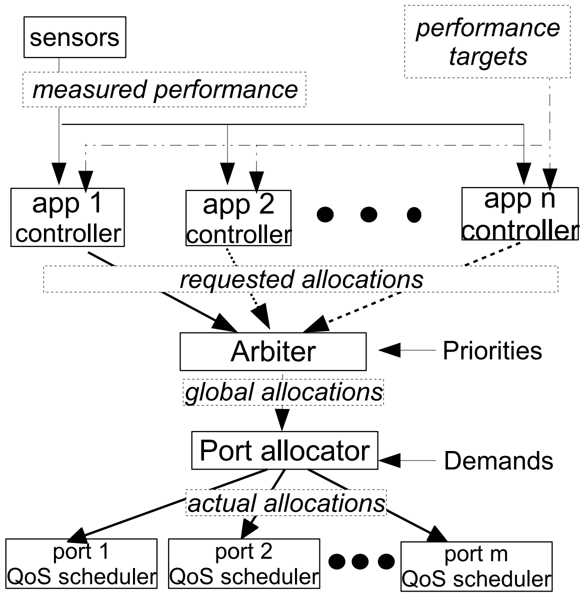


Figure 2: Architecture of the Storage QoS controller.

an administrative setting; we call this the *total concurrency bound*. We show that, by using the concurrency parameter, we can achieve per-application-class QoS targets across multiple ports even though each scheduler is independent.

We assume that an administrator specifies the QoS target for each application class. The specification includes a performance *metric*, a *target*, and a *priority*. The metric could be the desired throughput (IOs/sec), the bandwidth (bytes/sec), or the mean IO response time; other choices, such as the 90th percentile of the response time, are also possible. The target is the desired value of the metric specifying the desired performance. The priority provides an indication of how the application classes should be prioritized if the disk array cannot meet the targets of all the classes.

2.2 Storage QoS Controller

Our storage QoS controller consists of three layers, as shown in the Figure 2. The first layer is a set of *application controllers* that estimate the concurrency bound settings per application that will allow each application to reach its performance target. The second layer is an *arbiter*, which uses the application priorities with the concurrency requests and performance models generated by the application controllers to determine their global concurrency allocations. Finally, the *port allocator* determines the per-port concurrency settings for each application based on its global concurrency allocation and the recent distribution of its demands across the ports.

Application controller: Each application has a separate controller that computes the scheduler concurrency setting required to achieve its target. The application controller consists of two modules: a model estimator and a requirement estimator.

The model estimator module estimates a linear model for the dynamic relationship between the concurrency allocated to the application and its performance. Let $y_i(t)$ be the performance received by application i in control interval t, and $u_i(t)$ be the corresponding concurrency allocated to it. Then we use the approximation:

$$y_i(t) \approx y_i(t-1) + \beta_i(t)(u_i(t) - u_i(t-1))$$

The value of the slope $\beta_i(t)$ is re-estimated in every control interval using the past several measured values of application i's performance. These adjustments allow an application's model to incorporate implicitly the effects of the changing workload characteristics of all the applications (including itself). This linear estimation is designed to capture approximately the *local* behavior of the system, where the changes in the workload characteristics and the concurrencies allocated are small. These conditions apply because we re-estimate the model in every control interval (hence the workload characteristics do not change very much) and we constrain the controller to make only small changes to the concurrency allocations in each interval. We found empirically that the linear model performs reasonably well and provides an adequate approximation of the relationship between the concurrency allocations and the performance.

The requirement estimator module uses the model to compute how much concurrency the application requires to meet its target. This estimate is sent to the arbiter as the application's requested allocation. However, we limit the requested change from the previous allocation (by 5% of the total concurrency bound in our current implementation) to ensure that the system remains in a local operating region where the estimated linear model still applies. Also, the data from which the model is estimated is often noisy, and the resulting models can occasionally be quite inaccurate. Limiting the change in concurrency within a control cycle also limits the harm caused by an inaccurate model. The cost of this limit is that convergence to a new operating point is slowed down when application characteristics change, but we found rate of convergence adequate in empirical tests.

Arbiter: The arbiter computes the applications' actual global concurrency settings based on their priorities. In each control cycle, the arbiter receives the concurrency requests and the models used to derive them from each of the application controllers. There are two cases, the *underload* case, where the total concurrency bound is large enough to meet the independent requests submitted by the application controllers, and the *overload* case, where the total concurrency bound is smaller than the sum of the requests. In the case of underload, the scheduler parameters are set based on the application controllers' requests, and any excess concurrency available is distributed in proportion to the application priorities. In the overload case, the arbiter uses a linear optimization to find concurrency settings that will degrade each application's performance (relative to its target) in inverse proportion to its priority, as far as possible. As in the application controllers, we limit the deviation from the previous allocations so that the estimated linear model is applicable.

More precisely, say that there are n applications, p_i is the priority of application i, c_i its current allocation, u_i the next (future) allocation, and f_i is the linear model estimating the performance of application i in terms of u_i. l_i is the limit on deviation, which we set to 0.05 for all applications. The concurrency allocations c_i and u_i are normalized to the range $[0, 1]$ by dividing the actual allocation by the total concurrency bound. The performance values $f_i(u_i)$ are normalized to be (performance/target) for throughput and bandwidth metrics and (target/performance) for latency, in order to make all normalized performance values better as they increase. In order to compute the future allocations u_i, the arbiter solves the following linear program:

$$\begin{aligned}
&\text{Find } u_1, \ldots, u_n, \epsilon \text{ to minimize } \epsilon \text{ subject to:}\\
&\quad p_i(1 - f_i(u_i)) - p_j(1 - f_j(u_j)) < \epsilon\\
&\qquad\qquad\qquad\qquad \text{for } 1 \le i \ne j \le n\\
&\quad |u_i - c_i| \;\le\; l_i \qquad \text{for } 1 \le i \le n\\
&\quad u_1 + \cdots + u_n \;=\; \min(1, c_1 + l_1 + \cdots + c_n + l_n).
\end{aligned}$$

Note that the quantity in the right-hand side of the last constraint is a constant for the linear program, and the constraint is therefore still linear.

In the LP above, $p_i(1-f_i(u_i))$ is the fractional tracking error for application i, weighted by its priority. ϵ is the maximum difference between the priority-weighted fractional tracking errors for different applications, and the objective function tries to minimize this maximum difference. In the limit, $\epsilon = 0$, and the priority-weighted fractional tracking errors are equal; for example, in a scenario with two applications, if application 1 has the priority 1 and a performance value 10% below its target, and application 2 has the priority 2, then each has a priority-weighted tracking error of 0.1, and the performance value of application 2 should be 5% below its target.

Port allocator: The arbiter computes the aggregate concurrency setting for each application, but this concurrency has to be translated into per-port settings. Since application workloads may be dynamic and non-uniform across the ports, the port allocator uses the recently observed *demand* from each application at the ports to determine how much of the application's concurrency should be allocated to a port. We define an application's demand at a port as the mean number of IO requests outstanding from the application at the port during the previous control interval.

More precisely, let $d_{i,j}$ denote the the demand of application i through port j, and CG_i its aggregate concurrency as determined by the arbiter. Then the corresponding per-port normalized concurrencies are given by:

$$C_{i,j} = CG_i \left(\frac{d_{i,j}}{\sum_{k=1}^{n} d_{i,k}} \right)$$

where n is the number of ports. The normalized concurrencies are multiplied by the total concurrency bound and rounded up to determine the number of application IOs permitted to be scheduled simultaneously from that port. In addition, we set the concurrency setting to be at least one for all applications at all ports, in order to avoid blocking an application that begins sending IOs to a port during the control interval.

3. EVALUATION

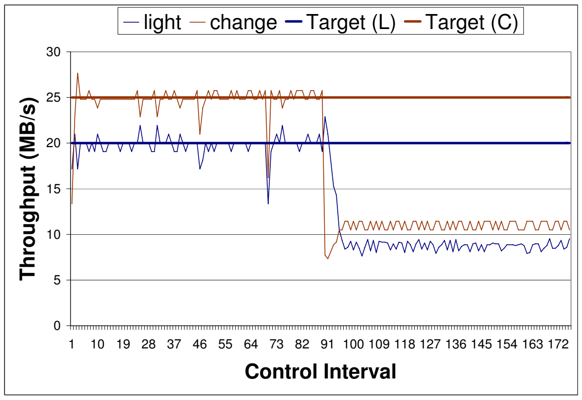
In this section, we present our initial results from a set of experiments we conducted to evaluate our storage controller. We designed our experiments to determine whether our controller can adjust the low-level scheduler parameters to achieve application targets. We also evaluated its ability to differentiate between multiple applications based on their priorities. Finally, we looked at the behavior of our controller when applications have different target metrics, for example a latency target and a throughput target.

We used two HP BL460c blade servers and a high-end XP-1024 disk array for our experiments. The blade servers are connected to separate ports of the XP-1024 disk array via two 4 Gbit/s QLogic Fibre channel adapters. Each of the blade servers had 8GB RAM, two 3GHz dual core Intel Xeon processors and used the Linux kernel version 2.6.18-8.el5 as their operating system. We allocated seven 4-disk RAID-1 logical disks for our experiments on the XP disk array.

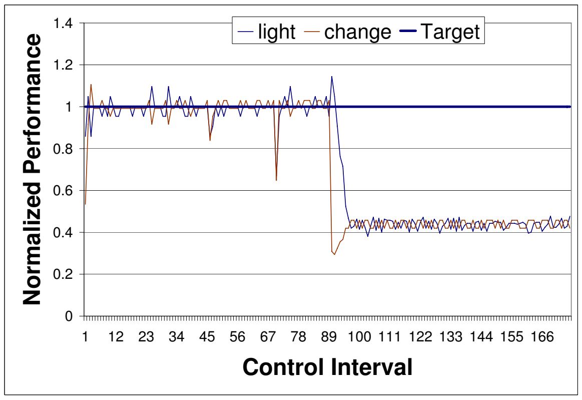
Since the XP array did not have an appropriate port scheduler that can limit available concurrency to specific workloads, we implemented a scheduler in the Linux kernel and ran it at each of the hosts to emulate independent port schedulers. The scheduler module creates pseudo devices (entries in /dev), which are then backed up by the logical disks at the XP array. Each pseudo device implements a different service level and we can associate different targets for these through our controller. The scheduler module intercepts the requests made to the pseudo devices and passes them to the XP array so long as the number of outstanding requests are less than the concurrency associated with the service level. In addition, the scheduler module collects the performance statistics needed by the controller. The controller polls the scheduler module at each control interval (every 2 seconds in our experiments) and gathers the statistics to determine overall performance levels achieved by all the applications classes.

We used a variety of synthetic workloads in our evaluation. Our reference workload is called `light`, and it consists of 16KB fixed size accesses generated by 25 independent threads. These accesses were made to locations selected at random. 90% of the accesses were reads and the 10% were writes. We also generated additional workloads based on `light` by varying the number of IO generating threads and the size of the IO requests in our evaluation.

Figure 3 presents the throughput of a `light` workload and a second, heavier workload with changing IO size, referred to as `change`. Initially, the `change` workload has 100 threads generating 16KB IOs. Midway through the experiment, the `change` workload starts sending small 4KB IOs instead. The targets for `light` and `change` are 20MB/s and 25MB/s respectively, and their priorities are equal. As the figure shows, both workloads initially meet their targets. However, when the request size of `change` drops, the it is unable to meet its target. Since the priorities of the two workloads are equal, the controller moves resources from `light` to `change`, so that the performance of both drops proportionately, which is the specified behavior. This is most clearly seen in Figure 3(b), which shows the throughputs of the two workloads normalized (divided) by their target values. The curves for the two



(a) Throughput



(b) Normalized Performance

Figure 3: Performance of a changing workload with throughput targets. The control interval is 2 seconds.

workloads are superposed, as we expect for equal priorities, and the performance of each workload goes from 100% of the target value to around 45% of the target value in the second half of the experiment.

In the second experiment, we used two workloads with different target metrics: the `light` workload with a throughput target of 25 MB/s and a different variant of the `change` workload with a latency target of 10ms. In this experiment, the `change` workload varies the number of IO generations threads in three phases; it uses 100 threads in the first phase, 25 threads in the second phase, and 5 threads in the third phase. Figure 4 presents the normalized performance of both of these workloads. In the first phase of the experiment, neither of the two workloads are able to meet their targets due to high intensity (100 threads) of the `change` workload. In the second phase, `change` reduces its intensity

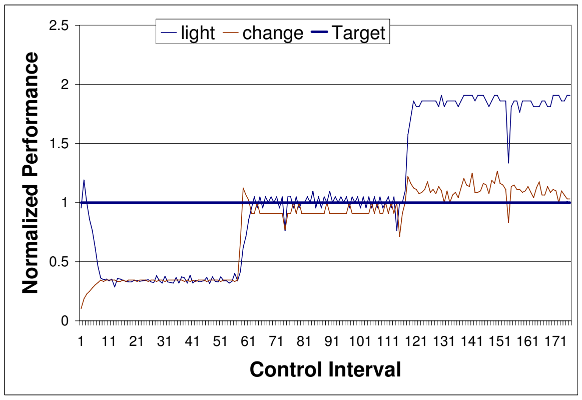


Figure 4: Workloads with different target metrics. The `light` workload has a throughput target of 20 MB/sec and the `heavy` workload has a latency target of 10ms.

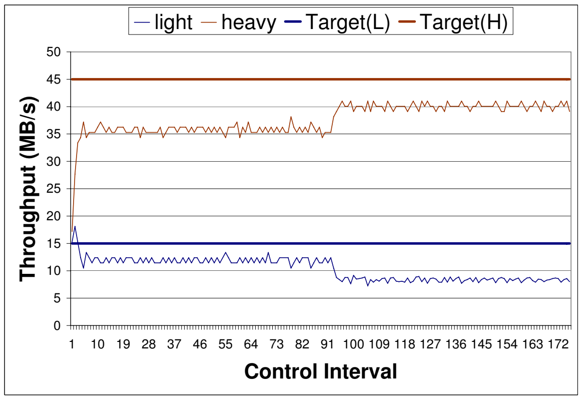
to 25 threads and as a result, both workloads are able to meet their respective targets: `light` achieves a throughput of about 25 MB/s and `change` experiences a reduction in its latency to about 10ms. Finally, in the last phase of the workload `change` reduces its intensity by using only five threads, and as a result, the `light` workload is able to further boost its throughput to about 38 MB/s without hurting the latency goals of the `change` workload. Note that, the `change` workload can not reduce its latency further as its requests are no longer queued (but instead dispatched directly); as a result the storage controller allocates the excess concurrency not used by the `change` workload to the `light` workload.

In the last experiment, we used two workloads `light` and `heavy`; the latter workload uses 100 threads to issue its requests. In this experiment, both `heavy` and `light` workloads start with equal priority in the first half of the experiment until the 90th control interval. Then, we adjusted the priority of the `heavy` workload to be four times that of the `light` workload. Figure 5 shows the effects of the priorities. It shows that the arbiter controller throttles the `light` workload so that the `heavy` workload is four times closer to its target compared to the `light` workload in the second phase of the experiment.

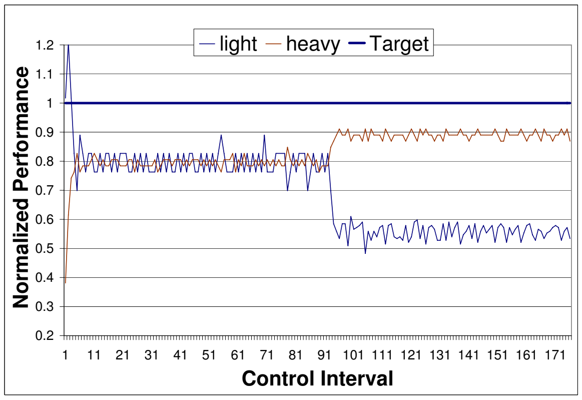
4. RELATED WORK

Prior work on controlling storage resources include systems that provide performance guarantees in storage systems [2, 4, 7, 12]. However, one has to tune these tools to achieve application-level guarantees. Our work builds on top of our earlier work, where we developed an adaptive controller [9] to achieve performance differentiation, and efficient adaptive proportional share scheduler [5] for storage systems.

In prior work, feedback controllers have been proposed [2, 9] to implement application prioritization requirements by using client throttling. A centralized controller monitors the state of the storage device and the application clients,



(a) Throughput



(b) Normalized Performance

Figure 5: Effects of workload priorities.

and then directs the clients to reduce or increase their IO request rate so that the requirements of the highest priority (or most demanding) client can be met. This approach can handle somewhat more complex prioritization requirements than a simple proportional share scheduler, but has the disadvantage that, by the time the clients implement the throttle commands, the state of the storage device may have changed; as a result, the controller may not be work-conserving, and the utilization of the storage device may be kept unnecessarily low.

In recent years, control theory has been applied to computer systems for resource management and performance control [6, 8]. Examples of its application include web server performance guarantees [1], dynamic adjustment of the cache size for multiple request classes [11], CPU and memory utilization control in web servers [3], adjustment of resource demands of virtual machines based on resource availability [15], and dynamic CPU allocations for multi-tier applications [10, 14]. In our prior work [13], we have developed a MIMO controller that can adjust the resource shares for multiple virtual machines to achieve application targets. In contrast, this work focuses on building a control system for a shared disk array with multiple ports. We also introduce the notion of resource control using dynamic concurrency bounds to provide performance differentiation across multiple ports.

5. CONCLUSIONS

In this paper, we presented a storage controller that dynamically allocates storage resources to multiple competing applications accessing data on a multi-port shared disk array. The controller consists of three layers, a set of application controllers, an arbiter and a port allocator. The application controllers determine the required I/O resources to meet application performance goals, and the arbiter uses application priorities to arbitrate in the case of overload. The port allocator uses each application's demands through a particular port to determine the per-port concurrency. Our preliminary experiments show that our controller can achieve application targets by automatically adjusting the scheduler parameters. The controller can also enforce different application priorities and different targets for multiple applications.

6. REFERENCES

[1] T. Abdelzaher, K. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13, 2002.

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

[3] Y. Diao, N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury. MIMO control of an apache web server: Modeling and controller design. In *Proc. of American Control Conference (ACC)*, 2002.

[4] P. Goyal, D. Modha, and R. Tewari. CacheCOW: providing QoS for storage system caches. In *Proc. of ACM SIGMETRICS*, pages 306–307, 2003.

[5] A. Gulati, A. Merchant, M. Uysal, and P. Varman. Efficient and adaptive proportional share I/O scheduling. Technical Report HPL-2007-186, HP Labs, Nov 2007.

[6] J. L. Hellerstein. Designing in control engineering of computing systems. In *Proc. of American Control Conference*, 2004.

[7] W. Jin, J. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proc. of ACM SIGMETRICS*, pages 37–48, 2004.

[8] C. Karamanolis, M. Karlsson, and X. Zhu. Designing controllable computer systems. In *Proc. of HOTOS*, pages 49–54, June 2005.

[9] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage*, 1(4):457–480, 2005.

[10] X. Liu, X. Zhu, P. Padala, Z. Wang, and S. Singhal. Optimal multivariate control for differentiated services on a shared hosting platform. In *Proc. of the IEEE Conference on Decision and Control (CDC)*, 2007.

[11] Y. Lu, T. Abdelzaher, and A. Saxena. Design, implementation, and evaluation of differentiated caching services. *IEEE Transactions on Parallel and Distributed Systems*, 15(5), May 2004.

[12] C. Lumb, A. Merchant, and G. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proc. of File and Storage Technologies (FAST)*. USENIX, 2003.

[13] P. Padala, K. Hou, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. G. Shin. Automated control of multiple virtualized resources. In *ACM Proc. of the EuroSys*, Mar. 2009.

[14] P. Padala, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, K. Salem, and K. G. Shin. Adaptive control of virutalized resources in utility computing environments. In *ACM Proc. of the EuroSys*, Mar. 2007.

[15] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In *Proc. of the Virtual Execution Environments, VEE*, pages 2–12, 2005.