Maestro: Quality-of-Service in Large Disk Arrays

Arif Merchant*
Google
aamerchant@google.com

Mustafa Uysal*
VMware
muysal@vmware.com

Pradeep Padala*
DOCOMO USA Labs
ppadala@docomolabs-usa.com

Xiaoyun Zhu*
VMware
xzhu@vmware.com

Sharad Singhal
HP Labs
sharad.singhal@hp.com

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

ABSTRACT

Provisioning storage in disk arrays is a difficult problem because many applications with different workload characteristics and priorities share resources provided by the array. Currently, storage in disk arrays is statically partitioned, leading to difficult choices between over-provisioning to meet peak demands and resource sharing to meet efficiency targets. In this paper, we present Maestro, a feedback controller that can manage resources on large disk arrays to provide performance differentiation among multiple applications. Maestro monitors the performance of each application and dynamically allocates the array resources so that diverse performance requirements can be met without static partitioning. It supports multiple performance metrics (e.g., latency and throughput) and application priorities so that important applications receive better performance in case of resource contention. By ensuring that high-priority applications sharing storage with other applications obtain the performance levels they require, Maestro makes it possible to use storage resources efficiently. We evaluate Maestro using both synthetic and real-world workloads on a large, commercial disk array. Our experiments indicate that Maestro can reliably adjust the allocation of disk array resources to achieve application performance targets.

1. INTRODUCTION

Consolidated storage environments typically have multiple applications store their data on shared, large disk arrays. For virtualized data centers, in particular, multiplexing application workloads on shared storage resources is the norm. This results in improved resource utilization, easier storage management, and lower cost. However, the applications using the shared storage present very different storage loads and have different performance requirements. For example, Online Transaction Processing (OLTP) applications typically present bursty loads and require bounded I/O response time; business analytics applications require high throughput; and back-up applications usually present intense, highly sequential workloads with high throughput requirements. In addition, some applications are more important from a business perspective and therefore have higher priority. It is critical to ensure that each application receives a performance commensurate with its needs and priority.

A shared storage system should ideally be able to provide each application with the performance it requires and keep one application from harming the performance of another. First, it should support performance differentiation between applications using per-application metrics and targets most appropriate to the applications. For example, one should be able to specify throughput targets for throughput-sensitive applications and latency targets for latency-sensitive applications. Second, since the applications can have different importance or urgency, the storage system should support prioritization among the applications, to be applied when the resources are not adequate to meet all the application requirements. For example, meeting the I/O response time requirement of an interactive system may take precedence over the throughput requirement of a backup system. Third, most large disk arrays have multiple I/O ports, and clients can be configured to access data through some or all of the ports. The performance differentiation and priorities should apply to the applications regardless of which ports they use. Finally, the mechanism must be simple, predictable, and robust, not requiring continual manual adjustment and tuning, even as workloads vary over time. Unfortunately, the methods reported to date in the literature are ineffective or do not support multiple application-specific performance metrics, explicit performance targets, and application priorities for overload conditions (see Section 2).

The main contribution of this paper is a new design, using adaptive feedback-control, for a storage QoS controller called Maestro that satisfies all of the requirements above. It supports application performance differentiation based on per-application metrics and targets, application priorities, and disk arrays with multiple ports. Since administrators need only to determine the requirements of individual applications and the priorities relative to the other applications, it is also simple to configure and use. Maestro monitors the performance of each application and periodically adjusts the allocation of I/O resources to applications to make sure that each application meets its performance target. If the resources available are insufficient to provide all the applications with their requested performance, Maestro sets the resource allocations so that each application's per-

*This work was done while the author was with HP Labs.

ICAC-11, June 14-18, 2011, Karlsruhe, Germany.

* Scheduler	Throughput target	Latency target	Overload priority	Performance insulation
SFQ(D)	Relative	No	No	No
Zygaria	Relative	No	No	No
AQuA	Relative	No	No	No
mClock	Relative	No	No	No
Avatar	Relative	Yes	No	No
PARDA	Relative	No	No	No
Triage	No	Yes	No	No
Façade	No	Yes	No	No
Argon	No	No	No	Yes
Fahrrad	No	No	No	Yes
Maestro	Absolute	Yes	Yes	No

Table 1: Comparison of storage scheduler features.

formance is reduced (relative to its target) in inverse proportion to its priority. We evaluated Maestro using an extensive set of synthetic benchmarks and trace workloads sharing a large commercial disk array. The results indicate that Maestro performs very well, maintaining application performance levels at or above the specified targets when there are adequate resources, and reducing the performance in accordance with each application's priority when there are inadequate resources.

The remainder of this paper is organized as follows. We describe the related work and how our contribution compares to it in Section 2. Section 3 explains the system model and the design of Maestro. Section 4 describes our prototype implementation. The experimental evaluation of Maestro is detailed in Section 5, and Section 6 presents our conclusions.

2. RELATED WORK

Existing QoS schedulers for shared storage can be divided into three categories: schedulers providing relative throughput allocations, schedulers providing per-application latency targets, and schedulers providing performance insulation.

Relative throughput: Proportional-share schedulers [**?**, **?**, **?**, **?**, **?**] provide relative throughput guarantees to active applications. These schedulers are mainly based on weighted fair queueing [**?**]. Typically, a weight is attached to each application, usually by the system administrator, and the available throughput from the storage device is shared between the active applications in proportion to the weights. Since storage throughput varies enormously depending upon the combination of workloads presented, absolute throughput targets cannot be supported by these schedulers. However, some schedulers, such as Zygaria [**?**] and AQuA [**?**], additionally support minimum throughput reservations based on a conservative estimate of the device throughput. mClock [**?**] supports both minimum throughput reservations and maximum throughput limits. PARDA [**?**] uses a single target for the device-level latency seen by all the hosts sharing the device and does not support per-application latency target. It uses a feedback control algorithm to tune the per-host maximum concurrency and employs proportional sharing to allocate individual application (VM) concurrencies within each host.

Latency targets: A few schedulers provide support for per-application I/O latency targets. Façade [**?**] enforces user-defined latency targets using the Earliest Deadline First (EDF) scheduler; it assumes that there is adequate I/O capacity to support all active workloads, and does not handle overload conditions. Triage [**?**] uses feedback-control to throttle all applications so that the most restrictive latency target is met. SLED [**?**] enforces per-application latency bounds by heuristically throttling applications receiving better than requested performance in favor of underserved applications. SARC-Avatar [**?**] straddles the relative-throughput and latency-target categories: it selects a group of I/Os to be scheduled using proportional sharing, but then orders those I/Os using the EDF policy to meet latency targets.

Performance insulation: Argon [**?**] provides each application with a fixed fraction of the performance it would receive if it had sole use of the device, by time-slicing the device appropriately and using cache partitioning. Fahrrad [**?**] allows applications to reserve a fixed fraction of a disk's utilization and enforces it efficiently via careful disk scheduling.

How Maestro is different: In discussions with storage system administrators, we found that, the properties of existing QoS schedulers were useful but not sufficient. Since the set of applications running on large arrays can be highly time-varying, it is not practical to use relative throughput allocations, since they would need to be adjusted very frequently. Performance insulation is extremely useful when the stand-alone performance of an application is known, but hard to use when initially provisioning an application. It is easier for administrators to specify absolute performance targets per application, since these do not change and are easy to reason about. Moreover, some applications have throughput requirements while others have latency requirements; thus, it is necessary to support both throughput and latency targets. Finally, while the load on arrays is arranged so that the targets can be met normally, the system must allow overload priorities that apply when not all performance targets can be met. The priority scheme needs to be flexible, to allow differing degrees of performance degradation under overload, without shutting out some workloads completely. Based on these requirements, we designed Maestro to support absolute throughput and latency targets of applications, as well a a flexible priority scheme. Table 1 compares the features of existing schedulers and Maestro.

Related techniques: Maestro uses adaptive feedback control to achieve the specified throughput and latency targets, and the desired prioritization under overload. Several researchers have applied control theory to computer systems for resource management and performance control [**?**, **?**]. Examples of its application include performance assurances for web servers [**?**], dynamic adjustment of the cache size for multiple request classes [**?**], CPU and memory utilization control in web servers [**?**], admission control for 3-tiered web sites [**?**], adjustment of resource demands of virtual machines based on resource availability [**?**], dynamic CPU allocations for multi-tier applications [**?**, **?**], and mitigation of performance interference among colocated cloud applications [**?**]. AutoControl [**?**] uses a MIMO controller to adjust resource shares for multiple virtual machines to achieve application performance targets. Our work builds upon techniques from this body of work.

3. STORAGE CONTROLLER DESIGN

This section first presents our system model, which is based on the typical design of large commercial disk arrays and their usage. It then describes the design of Maestro and details its components.

3.1 System model

Our system (Figure 1) consists of a disk array with a num-

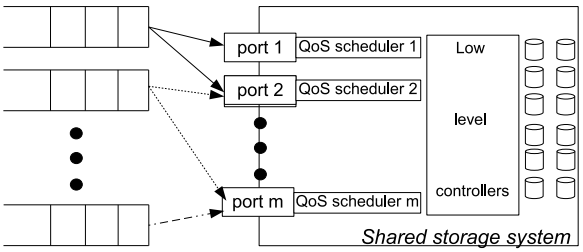


Figure 1: Storage system consists of a shared disk array with an independent proportional-share I/O scheduler running at each port.

ber of input ports where applications submit their I/O requests. Each application using the system has a specified performance target (either an I/O throughput or a latency) and a priority level. The performance target of the application is supplied by the application's user. Its priority level, which is relative to the other applications in the system, is specified by a system administrator; alternatively, the system administrator may assign a priority level to the user, which will apply to all his applications. The goal of Maestro is to enable each application to achieve its target performance or better when it is possible to do so. However, it is generally not possible to guarantee application performance levels without severely under-utilizing the array, because the performance depends so strongly upon the workload characteristics (such as temporal and spatial locality). When it is not possible to meet all the application performance targets, Maestro's goal is to deviate from the specified targets in inverse proportion to the application priorities. Note that this is a very flexible QoS framework, capable of emulating many other priority schemes; for example, one application can effectively be given absolute priority over another by making its priority level much higher.

The disk array allows control of the performance available to applications through QoS schedulers at the ports. Applications may submit their I/O requests to any subset of ports, in any proportion. The ports all share the same back-end resources. The QoS schedulers can be implemented in the disk array firmware or in an external "shim" device through which all I/O accesses pass. The port's QoS scheduler controls the resource sharing among the applications by limiting how many I/O requests each application can have outstanding at the disk array back-end from that port, and delaying some requests, if necessary. The number of I/O requests allowed outstanding is an adjustable parameter for each application and is set by an external controller. 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. In order to meet the QoS targets of the applications, an external feedback controller periodically polls the port I/O controllers to determine the performance each application is receiving and then adjusts the parameters of all the port QoS schedulers to meet the performance targets.

In our system, the port QoS schedulers have a single parameter, *concurrency bound*, for each application. The scheduler limits the number of I/O requests outstanding at the disk array back-end from each application to its concurrency bound. For example, if an application has a concurrency bound of 2, and it has 2 I/O requests pending at the back-end, the scheduler will not send any more requests from that application to the back-end until at least one of the pending requests finishes. The total concurrency

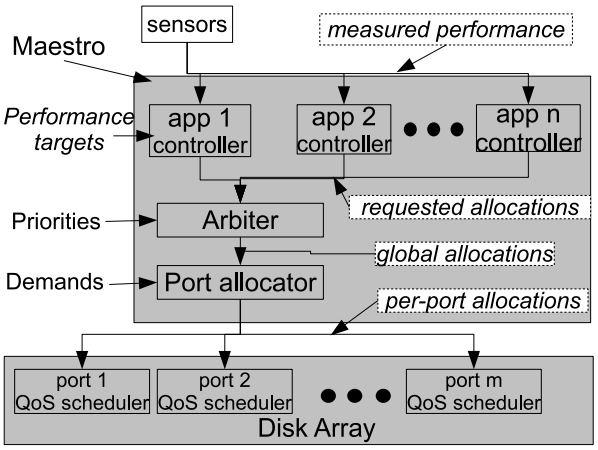


Figure 2: Architecture of Maestro.

of the array (i.e., the total number of I/Os permitted at the back-end from all ports) is limited, either by the system, or by an administrative setting; we call this constant the *total concurrency bound*, denoted by C. C is chosen by making a tradeoff between maximizing the array's throughput and making Maestro more responsive to workload changes. Allowing more simultaneous I/O requests into the array back-end means that there can be more I/O requests at each disk, which improves the disk throughput. On the other hand, because of the architecture of the array, once the requests are released to the array back-end, they proceed independently of Maestro. Hence, a large value of C limits the ability of Maestro to respond quickly to workload changes. Since we have found in the past that queueing 4 I/O requests per disk extracts most of the available throughput from the disk, we typically set C to 4 times the number of disks in the array.

3.2 The design of Maestro

Maestro consists of three layers that implements the QoS controller and uses the QoS schedulers to implement the controller's decisions, as shown in the Figure 2. The first layer is a set of *application controllers* that estimate the concurrency bound settings for each application to achieve its performance target. Application controllers also produce a model that estimates the relationship between the concurrency and the application performance. 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 setting for each application based on its global concurrency allocation and the recent distribution of its I/O requests across the ports.

3.2.1 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 dynamically estimates a linear model for the dynamic relationship between the concurrency allocated to the application and its performance. This linear estimation is designed to capture approximately the behavior of the system in the vicinity of the current operating point, where

SYMBOLS	DESCRIPTION
N	number of applications
M	number of ports
C	total permitted outstanding requests
a_i	the i^{th} application
p_i	priority
$y_i(t)$	normalized I/O performance
$U_i(t)$	concurrency limit
$\hat{U}_i(t)$	estimated concurrency requirement
$U_{i,j}(t)$	concurrency limit at port j
$d_{i,j}(t)$	demand at port j
c_i	current normalized concurrency
u_i	normalized concurrency for next interval
$\beta_i(t)$	slope of linear model for y_i
f_i	model of performance as function of u_i
l	limit on change of U_i in one control interval

Table 2: Notation used. All parameters with subscript i are for application a_i, and argument t means it applies to control time interval t.

the changes in the workload characteristics and the concurrencies allocated are small. The requirement estimator 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. Next, we describe the model and requirement estimator below in greater detail.

Model estimator: Suppose there are N applications, denoted as a_1, a_2, ..., a_N. (The notation we use in this paper is summarized in Table 2 for convenient reference.) Since the applications can have different performance metrics and targets, we first define a normalized performance metric that allows us to compare the performance of different applications in a uniform way. The normalized performance $y_i(t)$ received by application a_i in control interval t is defined as

$$y_i(t) = \begin{cases} \dfrac{\text{throughput for } a_i \text{ in interval } t}{\text{throughput target for } a_i} \\ \qquad\qquad \text{if } a_i \text{ uses a throughput metric;} \\ \dfrac{\text{latency target for } a_i}{\text{latency for } a_i \text{ in interval } t} \\ \qquad\qquad \text{if } a_i \text{ uses a latency metric.} \end{cases}$$

While it is primarily for notational convenience, this normalization has several other advantages: the normalized metric grows higher as the application performance improves, regardless of whether the performance metric is throughput or latency; the performance of different applications relative to their target values can be easily compared (e.g., $y_i = 0.9$ means that the application a_i is getting performance 10% below its target); and optimization with the normalized metric is more numerically stable.

Maestro requires a model to predict the effect of different concurrency allocations in the application performance. While there are a number of existing techniques for predicting storage performance [**?**, **?**], they are primarily focused on providing long-term, steady-state performance predictions, while we require predictions of how the performance will change immediately after a concurrency allocations changed. The transient performance varies quite non-linearly with the concurrency allocation, but it can be approximated locally with a linear model if the changes in allocation are small. We therefore use a linear auto-regressive model that is recomputed in each control interval.

Let $U_i(t)$ be the corresponding concurrency allocated to a_i in interval t, and recall that the total concurrency of the system is C. We then estimate a linear auto-regressive model for the normalized performance:

$$y_i(t) \approx y_i(t-1) + \beta_i(t)(U_i(t) - U_i(t-1))/C. \qquad (1)$$

The value of the slope $\beta_i(t)$ is re-estimated through linear regression in every control interval using the past several measured values of application a_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). Using a linear model is a reasonable approximation of the underlying system so long as the workload characteristics and the resource allocations do not change very much. These conditions apply because we re-estimate the model in every control interval, and hence, the workload characteristics generally do not change very much, and we constrain the controller to make only small changes to the concurrency allocations in each interval. We keep the duration of the control interval short, 2 seconds in our implementation, to frequently incorporate the changes in the workload conditions into the performance estimates. It is possible to use yet shorter control intervals, but we did not see any gain from doing so in our experiments. In actual use, we found that the linear model (1) captures the behavior of the system reasonably well when workload characteristics change slowly, but can be inaccurate if the workload changes abruptly. In addition, the application performance, which is used in the regression for the gradient $\beta_i(t)$, can vary due to many reasons and, as a result, the data used to measure $\beta_i(t)$ is noisy. To compensate for this, we monitor the linear regression used to estimate $\beta_i(t)$, and revert to the default model $\beta_i(t) = 1$ if the fit in the regression is poor, or if the value found for $\beta_i(t)$ is not within reasonable bounds (for example, if $\beta_i(t)$ is negative). The default model allows us to enforce the basic constraint that when a concurrency allocation for an application is increased, we expect a positive change in its normalized performance, and the slope of 1 gives a conservative prediction for the change in the performance metric. The application's performance in the next interval can be estimated by approximating $\beta_i(t+1) \approx \beta_i(t)$, which gives:

$$y_i(t+1) \approx y_i(t) + \beta_i(t)(U_i(t+1) - U_i(t))/C \qquad (2)$$

We also experimented with more sophisticated models, including higher-order auto-regressive models and quadratic models, but found that they were no more effective than this simple, linear, auto-regressive model.

Requirement estimator: The requirement estimator computes the concurrency required by the application, based on the model (2), and sends it to the arbiter as the application's requested allocation.

More specifically, the model (2) is used to compute the concurrency $\hat{U}_i(t+1)$ the application requires for the next interval (i.e., to make the normalized metric $y_i(t+1) >= 1$):

$$\hat{U}_i(t+1) = U_i(t) + C(1 - y_i(t))/\beta_i(t) \qquad (3)$$

This estimate is then modified, as described below, using two constraints, one to maintain high utilization and the other to limit the degree of change in the resource allocation.

In some situations, an application does not generate as many concurrent I/O requests as its requested allocation from the requirement estimator. This can happen, for example, if the number of I/O generating threads in the application is low or there are dependencies between successive I/O requests that limit the number of I/O requests the application can have pending at a time. If the application a_i used, on average, less than 80% of its allocated concurrency in the previous control interval, then the request estimator adjusts the value of $\hat{U}_i(t+1)$, based on the number of concurrent I/O requests issued in the previous interval, so that the average concurrency utilization of a_i is at least 80% in the next interval. This constraint ensures that the application is not allocated more concurrency than it can use, allowing the unused concurrency resource to be allocated to the other applications in the system.

The second constraint applied to the requested concurrency is that the change from the previous allocation is limited. This constraint can override the minimum utilization constraint, if needed. The limit is 5% in our current implementation; in other words, if necessary, we change $\hat{U}_i(t+1)$ to fit in the range $[U_i(t) - 0.05C, U_i(t) + 0.05C]$. This limit ensures that the system remains close to the current operating point, where the estimated linear model still applies. Also, the data from which the model is estimated are often noisy as workload patterns change, and the resulting models can occasionally be inaccurate until the workload behavior stabilizes. Limiting the change in concurrency within a control interval limits the harm caused by temporarily inaccurate models. 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.

3.2.2 Arbiter

The arbiter computes the applications' actual global concurrency settings for the next control interval based on their priorities. In each control interval, the arbiter receives the concurrency requests and the models used to derive them from each of the application controllers. There are two cases, (1) the *underload* case, where the total concurrency bound is large enough to meet the independent requests submitted by the application controllers, and (2) 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 applications' 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, the arbiter also limits the deviation from the previous allocations so that the estimated linear model is applicable.

More precisely, say, the arbiter is running at the end of control interval t and needs to compute the next concurrency allocation $U_i(t+1)$ for application a_i. We elide the underload case, since it is trivial. Suppose that the system is overloaded, or $\sum_j \hat{U}_j(t+1) > C$. We define $c_i = U_i(t)/C$ as the current normalized concurrency allocation to application a_i and $u_i = U_i(t+1)/C$ as its next normalized allocation. Let f_i be the linear model estimating the performance of application a_i in terms of u_i, based on Eq. (2): $f_i(u_i) = y_i(t) + \beta_i(t)(u_i - c_i)$. Let p_i be the priority of application a_i. In order to compute the future allocations u_i, the arbiter solves the following Linear Program (LP):

$$\begin{aligned} \textbf{Find } u_1, \ldots, u_N \text{ to } \textbf{minimize } \epsilon \textbf{ subject to:} \\ p_i(1 - f_i(u_i)) - p_j(1 - f_j(u_j)) &< \epsilon \\ &\text{for } 1 \le i \ne j \le n \\ |u_i - c_i| &\le l \quad \text{for } 1 \le i \le n \\ u_1 + \cdots + u_N &= 1. \end{aligned}$$

Note that the allocation for the previous control interval satisfies the constraints of this optimization, and hence it always has a feasible solution.

In the above LP, $p_i(1 - f_i(u_i))$ is the *priority-weighted fractional tracking error* (PWFTE) for application a_i, which measures how much below target its performance will be, weighted by its priority. ϵ is the maximum difference between the PWFTEs for different applications, and the objective function tries to minimize this maximum difference; note that ϵ is produced from the LP optimization, and is not user-supplied. In the limit, $\epsilon = 0$, all the PWFTEs are equal, and the fractional performance reduction of each application, relative to its target, is in inverse proportion to its priority. For example, consider a scenario with two applications, a_1, with priority $p_1 = 1$ and a_2, with priority $p_2 = 2$. If the performance of a_1 is 10% below its target, its PWFTE is 0.1. If $\epsilon = 0$, then a_2 has an equal PWFTE, and its performance should be $0.1/p_2$, or 5% below its target.

3.2.3 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 I/O requests outstanding from the application at the port during the previous control interval.

More precisely, let $d_{i,j}(t)$ denote the the demand of application a_i through port j during the current interval t, and $U_i(t+1) = C \cdot u_i$ its aggregate concurrency for the next interval as determined by the arbiter. Then, the corresponding per-port concurrencies are given by:

$$U_{i,j}(t+1) = U_i(t+1) \left(\frac{d_{i,j}(t)}{\sum_{k=1}^{M} d_{i,k}(t)} \right) \tag{4}$$

where M is the number of ports. These concurrency values are rounded up to determine the number of simultaneous application I/Os permitted from that port. In addition, we set the concurrency setting to be at least one for all applications at all ports, even if the applications do not access the port, in order to avoid blocking an application that begins sending I/Os to a port during the next control interval.

4. PROTOTYPE IMPLEMENTATION

Our prototype implementation of Maestro consists of two parts: a controller and a scheduler. The controller implements the three layers that are responsible for dynamically allocating the available concurrency in the disk array: application controllers, arbiter, and port allocator. The scheduler

implements the concurrency limiting mechanism that the controller uses to enforce its allocations. In addition, the scheduler collects the low-level performance statistics used by the controller.

We implemented the controller in Java as a user-level process in the Linux operating system. The scheduler is implemented in C as a Linux kernel module. The scheduler creates pseudo devices (entries in /dev), each representing a different service level. The scheduler module intercepts the requests made to the pseudo devices and passes them to the disk array so long as the number of outstanding requests are less than the concurrency limit. The scheduler also provides an `ioctl` interface for the controller to poll the performance statistics and set the concurrency limits of each application. The controller polls the scheduler at each control interval (every 2 seconds in our experiments) and gathers the statistics to determine overall performance levels achieved by all the applications. It then computes the concurrency allocations for the next interval and sets the concurrency limit for each pseudo device.

We used four HP BL460c blade servers and a high-end XP-1024 disk array for our experiments. The blade servers were connected to separate ports of the XP-1024 disk array via 4 Gbit/s QLogic Fibre channel adapters. Each server had 8GB RAM, two 3GHz dual-core Intel Xeon processors and used the Linux kernel version 2.6.18-8.el5 as its operating system. Each server ran the scheduler independently and the scheduler parameters were set once in every control interval. We ran the controller on one of the servers and used TCP sockets to communicate with the other schedulers.

We used 40 logical disk groups in the XP-1024, 29 of which were configured as RAID-5 and the remaining 11 as RAID-1. Each of the logical disk groups contained 4 disk drives. In total, we used 160 disk drives in our experimental setup. We created several logical volumes at each of the hosts using these disk groups: for the experiments with synthetic workloads (described in Section 5), we created a logical volume from 7 disk groups, and for the experiments with trace workloads, we created a logical volume out of 32 disk groups.

5. EVALUATION

We now evaluate Maestro using a variety of workloads. We first describe the workloads and then outline our evaluation methodology. Then, we present our results from the experiments we conducted. The experiments are divided into three categories: evaluating the functionality of Maestro, evaluating its robustness to different types of workloads and specifications, and testing with real-world workloads.

We used a variety of synthetic and trace-driven workloads in our experiments. Our reference synthetic workload, called `light`, consists of 16KB fixed size accesses generated by 25 independent threads. These accesses were made to locations selected at random. 90% of these accesses were reads and 10% were writes. We also generated additional workloads based on `light` by varying the number of I/O generating threads and the size of the I/O requests in our evaluation. The `heavy` workload used 100 threads with the same characteristics as `light`. The last synthetic workload is called `change`, where the workload starts as `heavy`, but later changes its characteristics as described in the experiments. It, too, uses a 90:10 mix of reads and writes.

We also used traces gathered from real-world applications in our evaluations by replaying the workloads in these traces on our XP-1024 disk array. We used two traces for this purpose: a trace of the TPC-C benchmark and a trace of an SAP system supporting more than 3000 users. Both of these are database workloads and the traces were obtained at the block I/O layer. The TPC-C workload was originally run on a mid-range HP EVA disk array; it consists of 60% reads of small (4 KB to 8 KB), random requests. The SAP benchmark originally used an XP-512 disk array with 16GB cache and 40 RAID groups with 4 disks each (160 disk drives). We replayed these traces on our XP-1024 disk array using 32 RAID groups of 4 disks each, spread across a total of 128 disk drives.

5.1 Evaluating functionality

Our first set of experiments are designed to validate the basic functionality of Maestro. We used synthetic workloads for these experiments. The three experiments test, respectively, that the Maestro achieves the desired performance differentiation, that it can simultaneously handle latency and throughput targets, and that it correctly differentiates performance based on the application priorities.

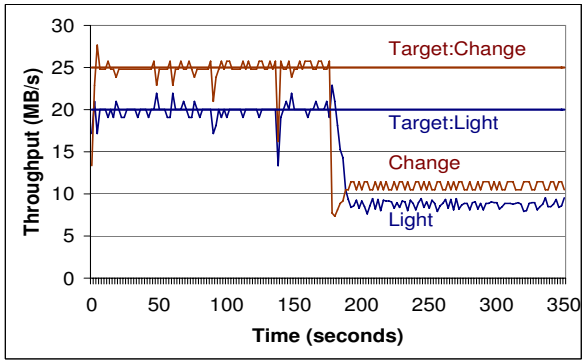


Figure 3: Performance differentiation.

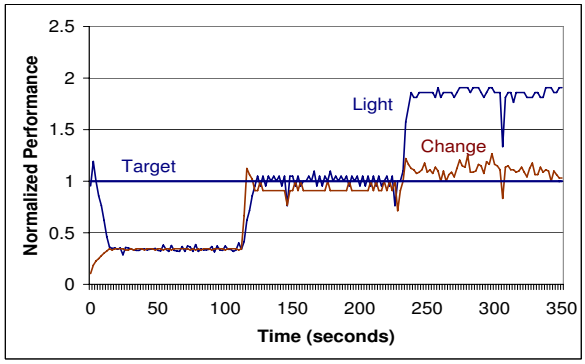


Figure 4: Workloads with different target metrics. The `light` workload has 20MB/s throughput target and the `change` workload has 10ms latency target.

Performance differentiation: To test that the Maestro achieves and maintains the desired performance differentiation, we ran an experiment with two equal-priority workloads with different throughput targets. The first workload is the previously described `light` workload, with a target of 20 MB/s. The second workload is `change`, a heavier workload with a changing I/O size, and a target of 25 MB/s. The `change` workload has 100 threads that initially generate 16KB I/Os and switch to small 4KB I/Os midway through

the experiment. Figure 3 plots the resulting throughputs. Both workloads initially meet their targets. However, when the request size of `change` drops, it is unable to meet its target, which now demands a higher I/O rate. Since the priorities of the two workloads are equal, Maestro moves resources from `light` to `change`, so that the performance of both drops proportionately, which is the specified behavior.

Different target metrics: In the second experiment, we used two equal-priority 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 I/O-generating 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 plots the normalized performance of both workloads: (throughput/target-throughput) for `light` and (target-latency/latency) for `change`. The target value for both workloads is 1, and higher is better; ideally, both should be at or above 1 and, since they have equal priorities, superposed when they are below 1. In the first phase of the experiment, neither of the two workloads is able to meet its target due to the high intensity (100 threads) of the `change` workload. In the second phase, `change` reduces its intensity 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 cannot reduce its latency further as its requests are no longer queued (but instead dispatched directly); as a result, Maestro allocates the excess concurrency not used by the `change` workload to the `light` workload.

Priorities: In the last experiment of this set, we used two workloads `light` and `heavy`; the latter workload uses 100 threads to issue I/O requests. We set the target for `light` as 15 MB/s and the `heavy` for 45 MB/s. In this experiment, both `heavy` and `light` workloads started with equal priorities. After 180 seconds, we adjusted the priority of the `heavy` workload to be four times that of the `light` workload. Figure 5 shows the effects of the priority change. The normalized throughputs of both workloads are initially equal, reflecting the equal priorities. When the priority of the `heavy` workload is increased, Maestro throttles the `light` workload so that the `heavy` workload is approximately four times closer to its target compared to the `light` workload, which, again, matches the specified priorities.

5.2 Evaluating robustness

We evaluated the behavior of Maestro in three aspects to determine its robustness: 1) with increasing numbers of workloads, 2) with bursty workloads, and 3) with workloads with non-uniform load distribution across the array ports.

Increasing workloads: To evaluate the effectiveness of the optimization-based resource allocation as the number of workloads increases, we ran experiments with 2, 4, 8, and 16 `light` workload instances sharing the disk array. In each case, we set the priorities of half of the workloads to 4 and half to 1. Figure 6 shows the normalized performance of the `high` priority workloads and the `low` priority workloads

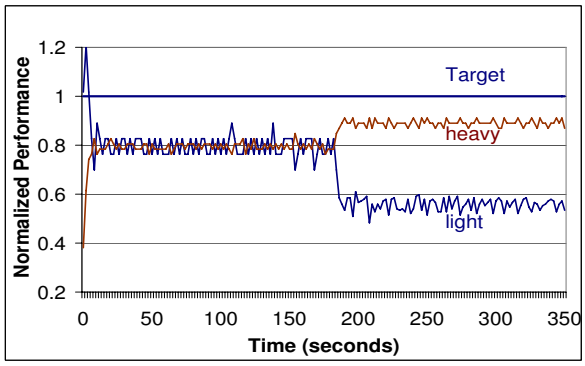


Figure 5: Effects of workload priorities.

for each of the workload mixes. The error bars indicate the standard deviation of the normalized performance during the experiment. Recall that, for the normalized performance, the performance of applications should either be above 1 (higher is better), or when it is below 1, the discrepancy for each application should be inversely proportional to the priority. As the figure shows, Maestro provides higher performance to high priority applications regardless of the number of workloads present. Second, the variation in the normalized performance generally stays low as the number of workloads increases. In the case of the 16-workload mix, the array is extremely overloaded, and Maestro allocates the per-port minimum resources allowed to the low-priority workloads. As a result, the performance of the `low` workloads is higher than their priorities would require. In the case of the 2-workload mix, the array is underloaded and the `high` workloads already get their maximum performance limit, and as a result, the 4:1 priority ratio does not lead to equivalent differentiation in performance. The `low` workloads had substantial variation in their performance. This is because they are allocated close to mimimum concurrency (1%, the minimum setting we used in this experiment), as a result, even a small change in the concurrency allocation causes the performance of the corresponding `low` workload to jump. For example, for a concurrency allocation of 1, increasing the allocation by one additional I/O concurrency is a 100% increase in allocated resources.

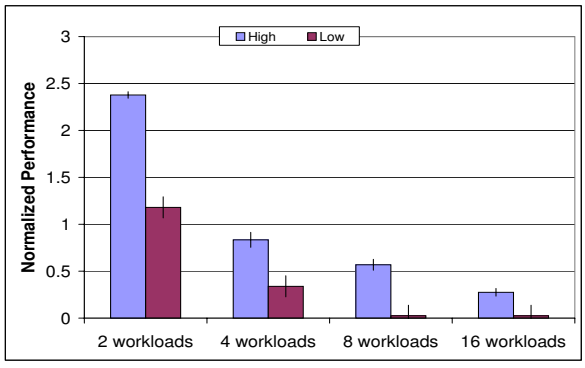


Figure 6: Increasing number of applications.

Bursty workloads: Next, we use a bursty workload to determine whether the controller allocations are robust to large peaks in workload demand. Figure 5.2 shows the normalized performance of the `light` workload sharing the ar-

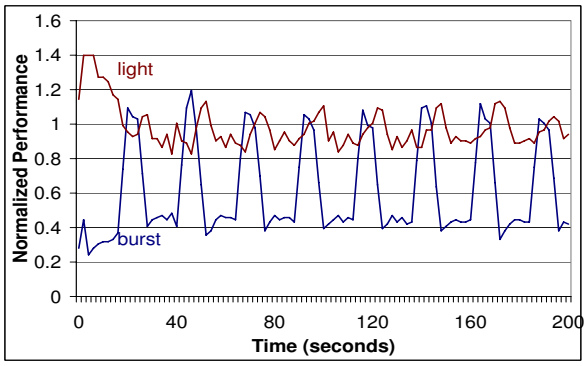


Figure 7: Impact of bursty workloads.

ray with a workload that has periodic bursts. The `light` workload has priority over the bursty workload (4:1). The bursty workload alternates between 50 and 200 I/O-generating threads every 20 seconds. Maestro is able to quickly change the concurrency allocation, closely following the peaks of the bursty workload while still keeping the performance of the high priority `light` workload close to its target.

Non-uniform workloads: Finally, we evaluate the impact of the port allocator in the presence of non-uniform, changing workloads across the multiple ports. For this experiment, we use 4 hosts and start 2 `light` workloads, `light1` and `light2`, at each host. Initially, each host generates equal load on its respective array port. After 140 seconds, the `light2` workload ceases generating requests at two of the ports, and simultaneously transfers the load to the other two ports. The overall load on the array is not changed. At 280 seconds after the experiment begins, the `light1` workload reallocates its workload similar to the `light2` workload. At this time, two ports receive all the requests, and two ports are entirely idle, but the overall workload is not changed. Figure 8 shows the fractional allocation of available concurrency (combined for both workloads) across the four ports during this experiment. In the first phase, when each port receives equal load, the concurrency allocation across ports is correspondingly equal. As the workload shifts its load from ports 1 and 3 to ports 0 and 2, the port allocator also shifts the allocated concurrency to these ports without any impact on the application performance.

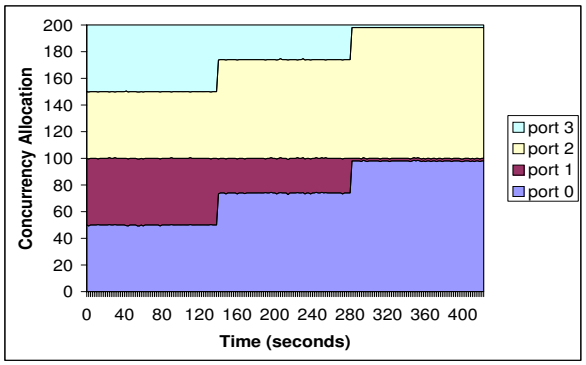
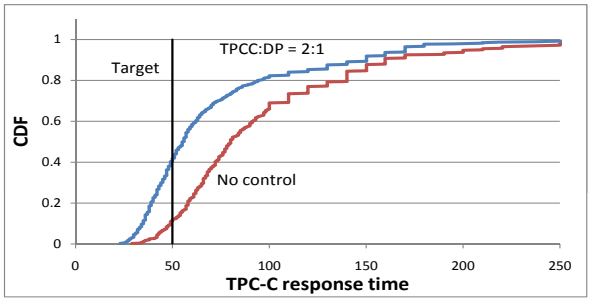


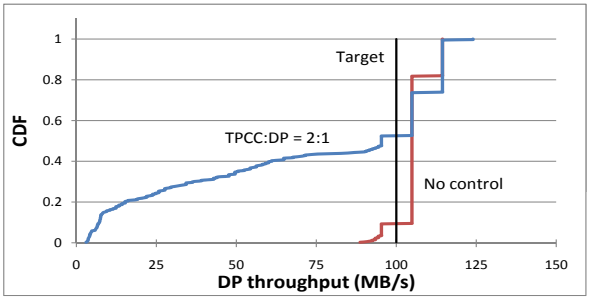
Figure 8: Changing workloads across multiple ports.

5.3 Real-World Workloads and Overload

In this section, we describe our results from a set of experiments that involve replaying traces gathered from real-world applications. To test the behavior of Maestro in overload conditions, we set targets that cannot all be met. We used two traces for this purpose: a trace of the TPC-C benchmark and a trace of a production SAP system supporting more than 3000 users. In our first set of experiments, we replayed these two traces with a background workload called `DP` using a workload generator with 200 independent threads generating 64K random reads. For each of the trace workloads, we set a response time target of 50ms. We used a throughput target for the background target for a sustained I/O rate of 100 MB/s.



(a) TPC-C



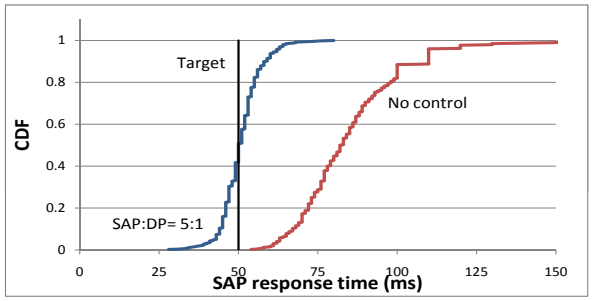
(b) DP

Figure 9: Replaying the TPC-C trace with a background workload on the XP disk array.

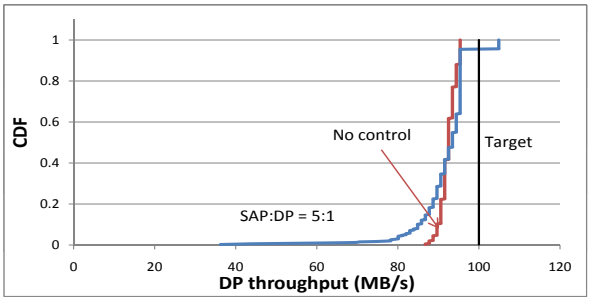
Figure 9 shows the distributions of the performance metrics of the TPC-C workload and the background workload, running with and without Maestro. The targets of 50ms latency for the TPC-C workload and 100 MB/s for the background workload were chosen to drive the storage system into overload, where it is not possible to meet both requirements. The TPC-C trace is a very bursty workload with periods of low utilization alternating with periods of high I/O intensity. We measured the latency of the TPC-C workload and the throughput of the background workload, averaged over 2-second intervals, and plotted the resulting cumulative distribution functions (CDFs). Without any control, the background workload achieved a maximum throughput of around 115 MB/s and its performance was above its target about 90% of time during the experiment. On the other hand, the TPC-C workload exceeded its target latency of 50ms about 90% of the time, with latencies in the range of 70ms to 120ms for 80% of the time during the experiment. This experiment shows that the high priority TPC-C work-

load was not able to meet its service-level objectives due to the interference from the heavy background workload.

We then ran the two workloads with Maestro, setting the priority of the TPC-C workload to be 2 times that of the background workload. We found that Maestro allocated more of the available concurrency to the higher priority TPC-C workload during periods of high I/O intensity. As a result, the TPC-C workload met its latency target of 50ms 42% of the time, as opposed to 10% of the time without control. The throughput of the background workload was still at or above its target for about half of the experiment, but Maestro favored TPC-C during periods of contention. Recall that there are not enough I/O resources for both workloads to meet their targets simultaneously all the time, and as a result, both workloads are unable to meet their targets for about half of the experiment duration. However, during periods of contention, the performance degradation of TPC-C was much lower than the performance degradation of the background workload.



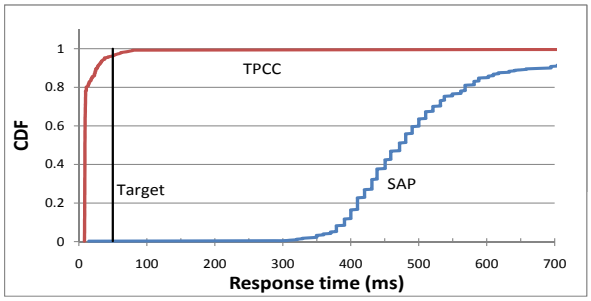
(a) SAP



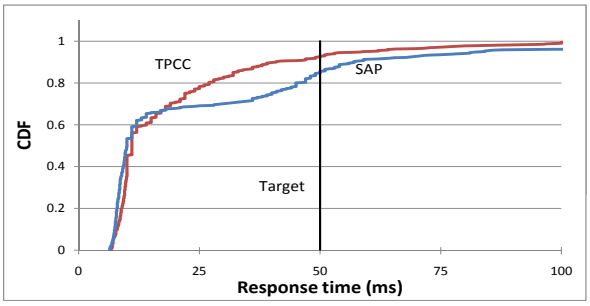
(b) DP

Figure 10: Replaying the SAP trace against a background workload on an XP disk array.

In Figure 10, we show the performance of the SAP workload and the DP workload, with and without Maestro. The SAP workload has a latency target of 50ms and the background workload has a throughput target of 100 MB/s. This is, again, an overload scenario, where storage system cannot give either SAP or the background workload enough performance to achieve its performance target. The background workload achieves about 90 MB/s throughput and the median latency of the SAP workload is around 80ms (with a long tail), well above its target. When we ran the same workload mix with Maestro, we set the priority of the SAP workload to be 5 times that of the DP workload. With Maestro enabled, the SAP workload received enough resources to satisfy its latency goal, while the background workload saw a small performance degradation (Figure 10(b)).



(a) No Controller



(b) Maestro

Figure 11: Replaying the SAP trace and the TPCC trace together on the XP disk array.

So far, we used trace workloads together with a synthetic background workload in our evaluation. We now show the TPC-C workload and the SAP workload running together to emulate the consolidated storage scenario where an OLTP application and a business analytics application share a disk array. Both applications have a response time target of 50ms. Figure 11(a) shows the performance of both applications running together, with no control, on the XP disk array. The performance of the SAP workload was heavily impacted by the presence of the TPC-C workload on the same array: with a median latency of 500 ms, it missed its latency target by an order of magnitude. On the other hand, the latency of the TPC-C workload was around 10ms most of the time, with occasional spikes up in the 30ms range, well below its 50ms target.

Figure 11(b) shows the performance of the SAP and the TPC-C workloads with Maestro. The priority of TPC-C was set to 2 and the priority of SAP was 1. Both applications met their performance targets for over 80% of the experiment duration. There was a slight degradation in the performance of the TPC-C workload compared with the uncontrolled case, but it still performed within the bounds of its target. In contrast, the performance of the SAP workload improved substantially, as Maestro gave it the resources to allow it to complete its I/Os quickly, despite the competing high-concurrency workload. Furthermore, the performance of the TPC-C workload remained strictly better than the

performance of the SAP workload, correctly reflecting its higher priority.

6. CONCLUSIONS

In this paper, we presented Maestro, a storage controller that dynamically allocates storage resources to multiple competing applications accessing data on a shared multi-port disk array. Maestro allows a user to specify the application's preferred metric (I/O throughput or latency), an explicit target value, and its priority. Maestro uses adaptive feedback to provide each application with its desired performance if possible. If the system is overloaded, and not all performance targets can be met, it reduces each application's performance, relative to its target, in inverse proportion to its priority. We argued that this is a simple, easily understood specification, which does not require users to know about the other applications sharing the storage system, unlike existing mechanisms that require considerable tuning. We presented an experimental evaluation of Maestro using both synthetic and real-world workload traces on a large, commercial disk array. Our experimental evaluation showed that Maestro can reliably achieve application performance requirements, and that it is robust in the face of diverse requirements, and variable, bursty workloads, even when the disk array is overloaded.