Triage: Performance Differentiation for Storage Systems Using Adaptive Control

MAGNUS KARLSSON, CHRISTOS KARAMANOLIS, and XIAOYUN ZHU
HP Labs, Palo Alto, CA

Ensuring performance isolation and differentiation among workloads that share a storage infrastructure is a basic requirement in consolidated data centers. Existing management tools rely on resource provisioning to meet performance goals; they require detailed knowledge of the system characteristics and the workloads. Provisioning is inherently slow to react to system and workload dynamics and, in the general case, it is not practical to provision for the worst case.

We propose a software-only solution that ensures predictable performance for storage access. It is applicable to a wide range of storage systems and makes no assumptions about workload characteristics. We use an online feedback loop with an adaptive controller that throttles storage access requests to ensure that the available system throughput is shared among workloads according to their performance goals and their relative importance. The controller considers the system as a "black box" and adapts automatically to system and workload changes. The controller is distributed to ensure high availability under overload conditions, and it can be used for both block and file access protocols. The evaluation of *Triage*, our experimental prototype, demonstrates workload isolation and differentiation in an overloaded cluster file-system where workloads and system components are changing.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; D.4.3 [**Operating Systems**]: File Systems Management; D.4.8 [**Operating Systems**]: Performance

General Terms: Performance, Management

Additional Key Words and Phrases: Storage resource management, policy-based management, QoS, clustered file systems, performance goals, performance management, control theory

1. INTRODUCTION

Resource consolidation in large data centers is a current trend across the IT industry and is mostly driven by economy-of-scale benefits. Consolidation is performed either within an enterprise or in hosting environments. In these data centers, storage systems are shared by workloads ofmultiple "customers". It is

A preliminary version of this article was published in *Proceedings of the International Workshop in Quality of Service*, 2004.
Authors' address: HP Labs, Hewlett Packard Company, 1501 Page Mill Road, Palo Alto, CA 94304; email: {magnus.karlsson,christos.karamanolis,xiaoyun.zhu}@hp.com.

important to ensure that customers receive the resources and performance they are entitled to. More specifically, the performance of a workload must be isolated from the activities of other workloads that share the same infrastructure. Furthermore, available resources should be shared among workloads according to their relative importance.

Existing state-of-the-art management tools rely on automatic provisioning of adequate resources to achieve certain performance goals [Anderson et al. 2002]. Although resource provisioning is necessary to meet the basic performance goals of workloads, it cannot handle rapid workload fluctuations and system changes. It is an inherently expensive and slow process—think of setting up servers, configuring logical volumes in disk arrays, or migrating data. Furthermore, it is too expensive to provision for the worst-case scenario, which is typically not known a priori. In our work, we ensure predictable performance of storage systems by arbitrating the use of existing resources under transient high-load conditions in a way that complements provisioning tools.

1.1 Resource Arbitration

In this article, we focus on storage system *throughput* as the key resource that is shared by the workloads. Throughput reflects the capacities of different physical resources in the system, such as server or controller utilization and network bandwidth. Throughput sharing is arbitrated by throttling storage access requests of different workloads. That is, requests from each workload are withheld somewhere on the data path and are released with a rate that complies with the targeted throughput for that workload.

The way to arbitrate the use of critical resources depends on the behavior of system components, their configuration, as well as workload dynamics. One way to achieve performance differentiation is to develop performance models that characterize the behavior of system components and workloads. Meeting the performance goals of the workloads while minimizing the overall use of resources is an optimization problem that can be solved using those models [Anderson et al. 2002]. Performance models that are developed offline have also been used in the context of feedback-based control loops [Abdelzaher et al. 2002; Diao et al. 2002a; Ko et al. 2003; Lu, et al. 2001; Robertsson et al. 2003, 2004]. However, enterprise-scale storage systems are large (with capacities often in the 100s of TBs), distributed, and increasingly heterogeneous, with constantly evolving hardware and software. Their workloads are complex, consisting of multiple overlapping I/O streams with unpredictable request patterns. Thus, it is impractical to devise detailed models of such systems offline to make performance predictions and control resource sharing.

1.2 A Control-Theoretic Approach

Because of the above observations, our approach is based on the assertion that the storage system must be considered as a “black box”. We assume no prior knowledge of the behavior of the system and its components, or the workloads applied to it, except that an increase in throughput generally results in higher request latencies. We solely depend on online performance monitoring from

outside the system to infer system models and perform workload throttling accordingly.

More specifically, we use an online feedback loop that includes a controller that makes throttling decisions based on the relationship between throughput and latency in the system. While the response latencies are within the specified goals for all workloads, the controller gradually increases the number of requests allowed to fully utilize the system. As soon as at least one workload's latency goal is violated, the controller starts throttling requests back in accordance with a specified resource sharing policy. In the context of such feedback-based control, it is important that we can argue about the properties of the resulting closed loop, namely that the overall system meets its goals (latency goals for each workload, while maximizing the utilization of the system) despite changes to the system and the workloads. To this aim, we use control theory to design the closed loop in a systematic way.

There is a number of existing systems that have applied control theory to computer systems: LotusNotes [Parekh et al. 2002], Apache [Abdelzaher et al. 2002, 2003; Abdelzaher and Bhatti 1999; Diao et al. 2002a; Lu, et al. 2001; Robertsson et al. 2003, 2004], Squid [Lu, et al. 2001], middleware [Li and Nahrstedt 1998], file server [Lee et al. 2004]. All these systems use *nonadaptive* controllers. The design of an nonadaptive controller depends on the assertion that a single (usually linear) model can be used to predict precisely the behavior of the controlled system. This model is developed offline and is then used to set parameters in the controller that are fixed over time. Those parameters specify how the controller reacts to feedback from the system. There are also several systems that use some form of feedback loop to control resource sharing, even though they do not use control theory in a formal sense [Chambliss et al. 2003; Diao et al. 2002b, 2002c, 2003a; Goyal et al. 2003; Lumb et al. 2003; Sundaram and Shenoy 2003; Welsh and Culler 2003]. All of them require some form of offline modeling and tuning of the system.

Storage systems exhibit a nonlinear behavior that depends on system configuration and workloads. For example, with a workload that retrieves data from an internal cache, the system exhibits a very different performance behavior compared to the case where data are retrieved from a disk. We show that, in the general case, it is not possible to design a well-behaved nonadaptive linear controller with parameters that are applicable to all different operating points of a storage system, because of the large variability in the system's operating range. The use of such a controller would result in long settling times or even instability[1] of the system when the operating point changes, as we report in Section 3. This precludes the use of any of the prior-art mentioned above.

To compensate for the lack of models that can be derived offline and to ensure that feedback-based control works in realistic systems with nonlinear behavior, we use *adaptive* controllers. Adaptive controllers do not depend on the existence of a single linear system model that needs to be derived offline. Instead, they periodically (the period is a design parameter) develop online a model of the

[1]The performance of different workloads would oscillate between extreme values without converging to the desired goals.

system at its current operating point. Based on that model, the controller parameters are adjusted dynamically so as to ensure stability and short settling times for the closed loop.

There is another recent case in the literature where an adaptive controller is used to provide differentiated web caching among competing workloads [Lu et al. 2002]. That work also aims at eliminating offline model development and controller tuning. However, in their case the goal is to provide proportional differentiation on content hit ratio, while in our work we target directly performance goals for each workload. In addition, our work aims at maximizing the utilization of existing system resources. Our work also differs in the design of the loop. First, we use a *direct self-tuning regulator*, a more computationally efficient type of adaptive controller that can be applied online with negligible overhead. Second, in the current prototype, *Triage*, workload control (throttling) is performed in a distributed fashion on the clients that use the service. Thus, differentiation can be achieved even with multiple storage servers, without requiring a central point of control. Moreover, our decentralized approach works effectively when the servers are overloaded, when differentiation is most important. The only other distributed control loop we are aware of [Stankovic et al. 2001] is based on a nonadaptive controller.

2. SPECIFYING PERFORMANCE OBJECTIVES

As discussed in section 1, this article proposes an online feedback loop that performs resource arbitration among workloads that compete for access to a shared storage infrastructure. This is done with two objectives. The first is to achieve *performance isolation* among the workloads. That is, a workload should obtain sufficient resources for the performance it is entitled to, irrespective of the behavior of other workloads in the system. Since it is not practical to provision the system sufficiently for the worst-case scenario, the second objective is to provide *performance differentiation* among workloads under overload conditions. In that case, resources should be shared among workloads on the basis of two criteria: (1) their relative importance; (2) the resources they already consume. We propose specifying two types of performance goals for each workload:

(1) A *latency goal* that should be met for all workload requests. This latency goal depends mostly on the characteristics of the corresponding application (timeouts, tolerance to delays, etc).
(2) A maximum *throughput allotment* for which the system should ensure isolation for the workload. This is the maximum throughput the customer is willing to "pay" for.[2]

These are both soft goals, that is, they refer to averages (or percentiles) over some time period. Further, we have to capture the relative importance of different workloads for the cases when the available system capacity cannot satisfy the maximum throughput allotments of all workloads. This can be accomplished by for example, having the users specify a utility function for each workload.

[2]Performance goal specifications for workloads are derived from high-level application goals or service level agreements. The way this mapping is performed is outside the scope of this article.

Table I. Example of Two Workloads Sharing the System According to Three Throughput Bands

	Band 0	Band 1	Band 2
aggr. throughput (IO/s)	0–100	100–400	400–900
workload W1	50%	100%	0%
workload W2	50%	0%	100%

Note: The top row shows the total system throughput in each band; the two rows below show the ratio by which the two workloads share that additional throughput.

The aim then is to differentiate the workloads so as to obtain the highest utility. In this article, we assume that somebody have already solved this optimization problem and provided us with the performance differentiation they would like to have between workloads. From typical utility functions, we observe that users do not usually assign the same importance to the entire range of throughput they require for their workloads. For example, the first few tens of IO/s are very important for the application to make progress. Above that, the value customers assign to the required throughput typically declines, but with different rates for various workloads. To capture the desired throughput differentiation between workloads, we specify a number of *bands* for the available system throughput. While these bands represent a piecewise function, any other function, such as a continuous one, could be used to specify the differentiation.

The details of how to specify workload throughput allotments can be best explained with an example. Consider a system with just two workloads. A business-critical workload W1 demands up to 350 IO/s, irrespective of other workload activities. Another workload W2 (e.g., one performing data mining) requires up to 550 IO/s. W2 is less important than W1, but it still requires at least 50 IO/s to make progress; otherwise the application may timeout and abort assuming that the storage system is not responding. That is also the case with W1, if it does not get 50 IO/s. To satisfy the combined throughput requirements of the two workloads, we specify three *bands* for throughput sharing, as shown in Table I. According to the specification, the first 100 IO/s in the system are shared equally between the two workloads, so that both can make progress. Any additional available throughput up to a total of 400 IO/s is reserved for W1. Thus, W1's 350 IO/s are met first. Any additional available throughput is given to W2 until its 550 IO/s goal is met. Any further throughput in the system is shared equally between the two workloads.

In general, any number of bands can be defined for any number of workloads that may share the system, following the principles of this example. If the system's capacity at some instance is sufficient to meet fully all throughput allotments up to band i, but not fully the allotments of band $i + 1$, then we say that the "system is operating in band $i + 1$". Any throughput above the sum of the throughputs of bands $0..i$ is shared among the workloads according to the ratios specified in band $i + 1$. The total available throughput indicates the "operating point" of the system. With 500 IO/s total system throughput in our example, the operating point of the system is 20% in band 2.

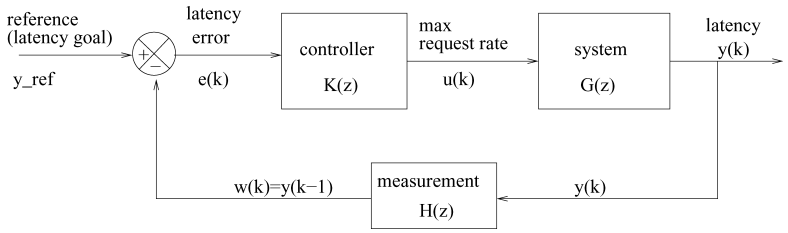


Fig. 1. Feedback loop with a nonadaptive controller for client request throttling.

In addition, the latency target of each workload should be met in the system. At some instance in time, the system is operating in a band i. As soon as the latency goal of at least one workload with a nonzero throughput allotment in any band $j, j \leq i$, is violated, the system must throttle the workloads back until no such violations are observed. Throttling within the specifications of band i may be sufficient, or the system may need to throttle more aggressively down to some band $k, k < i$. On the other hand, it is desirable to utilize the system's available throughput as much as possible. Therefore, when the system is operating in band i and the latency goals of all workloads with nonzero throughput allotments in bands $0..i$ are met, the system can let more requests through. This may result in the system operating in a band $m, m > i$.

3. DESIGNING A CONTROL LOOP

This section describes the design of the feedback loop for request throttling in the context of a client-server system that is typical of enterprise storage systems, irrespective of the storage access protocol used. The system consists of a number of storage servers and a number of client nodes that access data on the servers. One or more workloads may originate from a client, or more than one clients can generate a workload. To keep the discussion simple, we assume here that there is a 1:1 mapping between clients and workloads. Examples of such systems include network file systems [Callaghan et al. 1995], cluster file systems [Lustre 2005], or block-based storage [Saito et al. 2004]. For the discussions in this article, we use an installation of a cluster file system, Lustre [2005], with eight clients and one or two servers.

The objective is to design a feedback controller that arbitrates the use of system throughput by throttling client requests according to the specifications of the throughput bands and the latency goals. Since we follow a black-box approach as far as the behavior of the system is concerned, we require that the feedback loop depends merely on *externally observed metrics* of the system's performance, that is, response latency and throughput.

Figure 1 shows an abstract representation of the feedback loop. In the figure, $w(k)$ is the observed latency of the system at time k averaged over some sampling period, the length of which is specified by the system identification process in Section 3.1. k signifies what sample instance this is, with $k-1$ being the previous sample instance, k the current, and $k+1$ the next sample instance in the future. The input to the closed-loop system, y_{ref}, is the reference value (desired target value) for latency $w(k)$. Based on the difference between $w(k)$ and y_{ref},

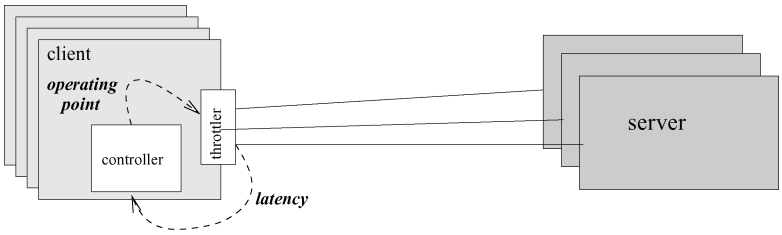


Fig. 2. A controlled Lustre instantiation.

the controller *actuates* the system by setting the operating point $u(k)$. This is the maximum aggregate throughput allowed to be obtained from the system (for all workloads); how this throughput is shared among workloads is determined according to the specified bands. Enforcing this maximum throughput requires that a throttling module intercepts requests somewhere on the datapath—it could be either on the clients or somewhere on the network. No assumption is made about the exact location of the controller itself. However, from a design perspective, it is desirable that:

(1) the controller reacts to end-to-end latencies as perceived by the application, since these capture overall system capacity, including for example storage area network bandwidth;
(2) the controller is designed in a decentralized way to ensure it is highly available even in an overloaded system, which is exactly what the feedback loop is designed to address.

Workload throttling can be performed (as determined by the controller) at any point on the datapath between clients and servers. It could be on the servers, on the clients, or somewhere in between (e.g., some intelligent switch), depending on what entity in the system can be modified and controlled. Being able to throttle on or close to the clients has the practical advantage that performance differentiation can also deal with contention in the network. So, for Triage, the prototype we discuss in this article, we assume that we have clients that can be modified to accommodate a throttling module as depicted in Figure 2. We believe this to be a realistic assumption given that we focus on data centers, where the software of every node can be configured as necessary. Moreover, there is a feedback loop with a separate controller for each client/workload in the system. The reference input to a controller is the latency goal for the that client's workload and the error is estimated locally. The controller calculates locally the operating point of the system, from its own perspective. The corresponding share for the local workload is derived from the throughput bands specification—all clients know that table. This does not create any strict synchronization requirements among clients, as this table changes infrequently. The controller modules in the different clients have to agree on the lowest operating point, as this is used across all clients. (If the minimal value was not used, some clients might send too many requests and violate isolation.) This requires some simple agreement protocol among the clients, that is executed once every sampling period. For example, a specific client (e.g., the one with the smallest id) calculates the

operating point locally and sends it to all other clients; other clients respond to the group only if they have calculated a lower value than that (the details of such a protocol are outside the scope of this article). The throttler imposes a maximum request rate for outgoing requests from the corresponding client.

3.1 System Identification and Validation

In this section, we describe the design of a feedback control loop using a control-theoretic nonadaptive controller. We do this for four reasons. First, we show that nonadaptive controllers are inadequate for storage systems and their workloads. Second, we use them as a comparison baseline for the adaptive controller we propose as a solution to our problem. Third, the offline system identification technique described here forms the basis for the online estimation technique used for the adaptive controller. Fourth, we derive some properties of the system which are also used for the design of the adaptive closed loop.

The first step toward designing a nonadaptive controller is to develop a model of the target system. Once we have that model, we can use control theoretic methods to design a controller for the system so that the resulting closed loop is stable and converges fast to the performance goal. Once the controller is designed, the model is not used anymore in the case of a nonadaptive controller. One type of model most frequently used to design controllers is the following linear model [Franklin et al. 1998; Hellerstein et al. 2004]:

$$y(k) = \sum_{i=1}^{N} \alpha_i y(k-i) + \sum_{i=0}^{M} \beta_i u(k-i). \tag{1}$$

In this model, $y(k)$ is the real latency[3] of the requests at time k and $u(k)$ is the operating point set at time k. The numbers N and M are the *order* of the system model, which captures the extent of correlation between the system's current and past states. Usually $N \geq M$ and for simplicity, we will set $N = M$ for the rest of this section. α_i and β_i are unknown model parameters. Thus, the requirement is to find values for α_i, β_i, N, M and a sample period that will produce a model that represents the system accurately. The process of determining these values is called *system identification*.

An important aspect of system identification is to find the order of the model (N) that results in a good fit for the measured data. This is related to the sampling period used to obtain measurements of the system and the inertia or "memory" of the system. When the request latency is much smaller than the sampling period, a first-order model ($N = 1$) is usually sufficient to capture the dynamics of the system, as there are few requests that affect the system across two consecutive intervals. Thus, requests occurring at time $k - 2$ or earlier have little impact on the latencies at time k. If, however, request latencies are comparable to (or longer than) the sampling period, then higher order system models are required. Intuitively, a long sampling period may result in slow

[3]The reason we use $y(k)$ here instead of $w(k)$ is that we need to have a model of the real effect of the actuation on the system, not the measured effect of the actuation. In general, deriving a controller from a system model made out of measured data ($w(k)$) might lead to instabilities. In our specific case, it would lead to a slower controller.

Table II. R^2 Fit of a First-Order Model and Residual Correlation Coefficient as a Function of Sample Interval

Model of	Sample Interval (ms)									
	1000		750		500		300		100	
	R^2	C_{coef}	R^2	C_{coef}	R^2	C_{coef}	R^2	C_{coef}	R^2	C_{coef}
Cache	0.764	0.04	0.745	0.05	0.685	0.04	0.479	0.140	0.439	0.140
Disk	0.416	0.045	0.399	0.05	0.379	0.03	0.159	0.047	0.112	0.026

Note: Two workloads: (i) all accesses in the cache; (ii) all accesses on random locations on disk.

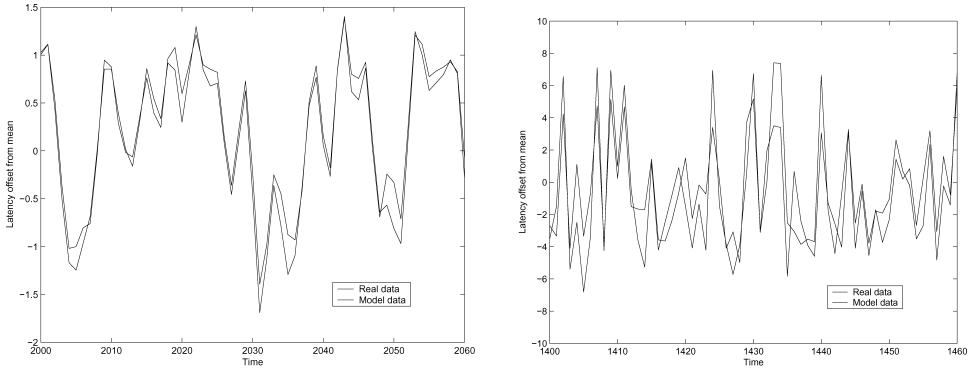
reaction and thus insufficient actuation by the controller. On the other hand, a short sampling period may result in considerable measurement noise and model over-fitting [Franklin et al. 1998], which in turn can lead to oscillations in the controlled system.

Since we consider the system as a black box, we have to use statistical methods to obtain the model (as opposed, e.g., to analytical queuing theory techniques). In order for these methods to work reliably, we need a set of measurements that satisfy the following two criteria. First, the measurements should be derived from actuation that spans the whole range of possible $u(k)$ values. Second, the measurements should reflect all possible deltas between consecutive actuations $u(k)$ and $u(k-1)$. Large deltas might affect the system differently than small ones. For large number of measurements, one method that satisfies the above criteria is to select a $u(k)$ at each sample period drawn from a uniformly random distribution covering the whole set of possible $u(k)$ values. In other words, $u(k)$ is a white noise signal consisting of signals that cover the entire spectrum of potential input frequencies. We then fit the obtained measurements to (1) by using least-squares regression (LSR)[4] [Ljung 1998].

We implemented this identification process in the throttler. The clients send as many requests as they are allowed to by the throttler. To ensure that worst-case system dynamics are captured by the identification process, we use the maximum number of clients (eight in our system) and look at the performance of two extreme workload cases: (i) the entire data set fits in the servers' cache (DRAM memory); (ii) all requests go to random locations on the servers' disks. Most workloads fall somewhere between these two extremes.

Table II shows the R^2 fit and the correlation coefficient of the residual error for a first-order model as the sampling period is varied. They are both model-fitting metrics. The *correlation coefficient of the residual error* [Franklin et al. 1998] is a number between 0 and 1 (the lower the better), which shows how much predictable information from the measured data is not captured by the model. A value close to zero means that there is no more predictable information in the data for us to extract. The R^2 *fit* [Franklin et al. 1998] is also a number between 0 and 1 (the higher the better), that indicates how much variation in the measured data is represented by the model. In the table, R^2 values are worse for the on-disk model, because measured latencies are more unpredictable in that case.

[4]At a high level, LSR is based on the assumption that large measurement changes are highly unlikely to be caused by noise and thus should be taken into account more than small changes.



$y(k) = 0.003827u(k) + 0.04554y(k-1)$
(i) All accesses in cache.

$y(k) = 0.1377u(k) + 0.001109y(k-1)$
(ii) All accesses on disk.

Fig. 3. System identification. Two extreme cases: the entire data set in cache and on disk, respectively.

In Table II, we observe that a first-order model extracts most of the information from the data. The two exceptions are the 300 and 100 ms intervals for the in-cache case. We have tried higher-order models for these cases, but they resulted in less than 0.05 improvement to R^2 fits—they are still a lot worse than having a sample period $\geq$ 500 ms. Thus, we use *first-order models* ($N = 1$) for the rest of the article. We also see that sampling intervals of 500 ms or higher provide the best fits. As 500 ms is close to the sample period where the model degrades, we pick a sample period of 1 s for the rest of the article.

Note that traditionally $R^2 \geq 0.8$ is considered to be a good fit for a system that can be approximated by a linear equation. As this is not the case with our system, we have to look at a plot of the model data versus the real data to judge whether the model is good. Figure 3 shows that both models predict the trends correctly, but miss the correct magnitude of the value in extreme cases, a situation that R^2 is biased against.

The captions in Figure 3 show the *two models* we estimated for the two extreme cases of workloads. The two models are substantially different, which, as we will see, results in different controller designs for each case. Also, in both models, the latency at time k, $y(k)$, depends heavily on the actuation of the system, $u(k)$, at the same time. The reaction of our system to the actuation is instantaneous. The intuition behind this is that our sample period is orders of magnitude larger than the request latencies. Thus, in the models, $y(k)$ depends on $u(k)$ rather than on $u(k-1)$. Also, $y(k)$ depends more on the actuation setting and much less on the latency at time $k-1$.[5]

3.2 Nonadaptive Controller Design

Having completed system identification, the next step is to design and assess a controller for the feedback loop of Figure 1. It is known from the literature

[5]This is so, because the value of $y(k)$ is typically in the range of 10^{-1}, while $u(k)$ is in the range of 10^2.

that for a first-order system like ours, a simple *integral* (I) controller suffices to control the system [Franklin et al. 1998]. The following is the time-domain difference equation for an I-controller:

$$u(k) = u(k-1) + K_I e(k) \tag{2}$$

In our system, $e(k) = y_{ref} - w(k)$, where $w(k)$ is the average measured latency in the system at time k. This average measurement contains request latencies that were produced by the system between time $k-1$ and k. In the worst case, $w(k)$ would be based solely on latencies measured close to time $k-1$. Thus we set $w(k) = y(k-1)$, so that our analytical arguments hold even under those circumstances. The controller output, $u(k)$, is the desirable operating point of the system at time k. K_I is a constant controller parameter that captures how reactive the controller is to the observed error. An integral controller ensures that the measured error in the system output goes to zero in steady state if the reference signal is a step function [Franklin et al. 1998]. For our system, this means that the system latency will be able to track the latency reference in steady state. Intuitively, when the latency error is positive (i.e., the measured latency is lower than the reference), $u(k)$ is larger than $u(k-1)$ to allow more requests to go through to fully utilize the system. On the other hand, a negative latency error indicates an overload condition in the system, and the controller decreases $u(k)$ to throttle back the requests to meet the latency goals.

We can now use control theory to come up with a value for K_I that leads to a stable system with low settling times and low overshoot. In order to accomplish this, it is usually convenient to perform the analysis in the Z-domain [Franklin et al. 1998; Hellerstein et al. 2004]. The Z-domain is a domain in which time series such as the controller (2) and the model (1) can be expressed as the sum of powers of the variable z. z can be viewed as a time shift operator, where z^{-1} means a delay of one time step, while z^2 signifies two time steps in the future. Control theory can then be used to design the closed loop by analyzing the *transfer function* of the closed-loop system. This is the function in the Z-domain that relates a change in the input to the controller (either from a change in the reference signal or the measured latency) to the effect on the output when the system is in a closed loop. To do this, we first need the Z-transform of all the components of our loop. The Z-transform of the controller, $K(z)$, can be derived from (2) as follows:

$$U(z) = U(z)z^{-1} + K_I E(z) \Rightarrow K(z) = \frac{U(z)}{E(z)} = \frac{zK_I}{z-1}. \tag{3}$$

$U(z)$, $Y(z)$ and $E(z)$ are the equivalents of variables $u(k)$, $y(k)$ and $e(k)$ respectively in the Z-domain. Using the same technique on the system models, we find that $G(z) = G_c(z) = \frac{0.003827z}{z-0.04554}$ or $G(z) = G_d(z) = \frac{0.1377z}{z-0.001109}$ respectively, for each of the two system models of Figure 3. $H(z) = z^{-1}$ represents a delay of one interval for the real system latency to be observed by the controller due to averaging over a sampling period.

The transfer function of the closed-loop system, $T(z)$, can be derived from the Z-transforms of its components, shown in Figure 1, using standard algebraic

manipulations of Z-transforms [Franklin et al. 1998]. It is as follows:

$$T(z) = \frac{Y(z)}{Y_{ref}(z)} = \frac{K(z)G(z)}{1 + K(z)G(z)H(z)} = \frac{N(z)}{D(z)}. \quad (4)$$

The solutions to equation $D(z) = 0$ are called the *poles* of the transfer function while the solutions to $N(z) = 0$ are called the *zeroes*. As will be explained later, the poles indicate whether the closed loop is stable or not, while the zeroes affect the settling time and overshoot of the controller. Inserting the values for $G(z)$, $K(z)$ and $H(z)$ above into (4), we obtain two versions of the system's transfer function, one for each of the system models:

$$T_c(z) = \frac{0.003827K_I z^3}{z(z^2 - (1.046 - 0.003827K_I)z + 0.04554)}, \quad (5)$$

$$T_d(z) = \frac{0.1377K_I z^3}{z(z^2 - (1 - 0.1377K_I)z + 0.001109)}. \quad (6)$$

Control theory states that if the poles of $T(z)$ are within the unit circle ($|z| < 1$ for all z such that $D(z) = 0$), the system is stable. Both transfer functions have a denominator $D(z)$, which is a third-order polynomial. However, one of the poles is always at zero. Thus, we can only control the location of two poles. Solving this, we find that the system is stable with $0 < K_{Ic} \leq 546$ for the on-cache workload, and with $0 < K_{Id} \leq 14.5$ for the on-disk workload. It is very important that the system is stable irrespective of whether the data is retrieved from the cache or from the disk, as this depends not only on the access pattern of the workload but also on other workloads' activities in the system. For example, one workload might have its data completely in the cache if it is running alone in the system, but it might have all its data being retrieved from the disk if there are other concurrent workloads evicting its data from the cache. This means that only for values $0 < K_I \leq 14.5$, the closed loop system is stable in practice.

However, stability alone is not enough. We need to pick a value for K_I that also results in low settling times and low overshoot, for the entire range of possible system models. To do this, we use the transfer functions to calculate the output values of the system under a step excitation, for different values of K_I. As Figure 4 shows, $K_{Ic} = 213$ and $K_{Id} = 7.2$ are good values for the in-cache and on-disk models, respectively. However, there is no single K_I value that could work for both cases. Indeed, as Figure 5 shows, when a controller designed for the in-cache model (with $K_{Ic} = 213$) is applied to a system with most accesses on disk, it results to an unstable closed loop system. Conversely, when a controller designed for the disk model (with $K_{Id} = 7.2$) is applied to a workload that mostly retrieves data from the cache, we end up with unacceptably long settling times and oscillations.

In conclusion, nonadaptive controllers designed following an offline system identification and design approach, as described above, do not work with storage systems like those we study here. Storage systems typically change a lot, either because of the dynamics of multiple concurrent workloads or because of changes in the configuration of the system itself. The latter may happen either because of failures of system components or because of management actions such as

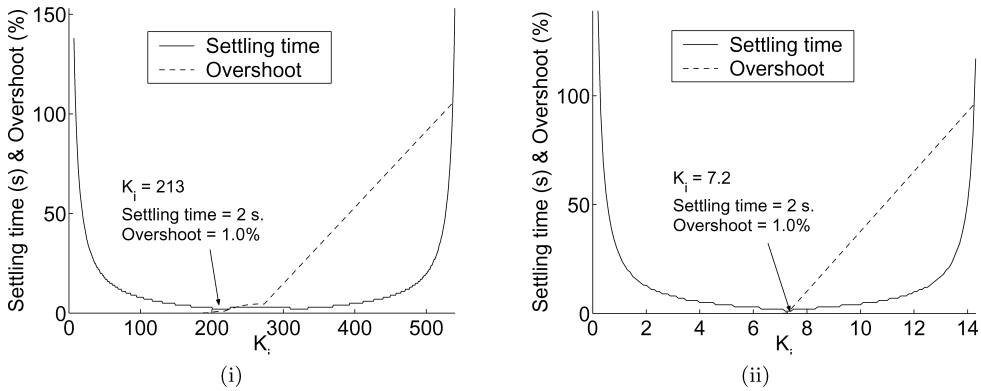


Fig. 4. The settling time and overshoot as a function of K_i when all data is in the cache (i) and on the disk (ii). Both diagrams have two y-axes: settling time (s) and overshoot (%).

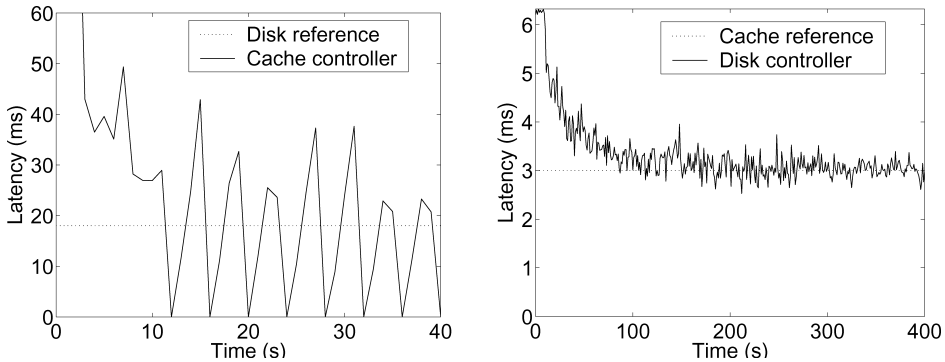


Fig. 5. The performance of the nonadaptive controller. A controller designed for the in-cache model is applied to a mostly on-disk workloads (left). A controller for the on-disk model is applied to a mostly in-cache workload (right). A latency of 0 means that there were no requests during that sample period.

(re)provisioning. We cannot even design a separate nonadaptive controller for each possible operating point of the system, because such operating points are not known a priori, for realistic storage systems. In this section, we analyzed the feasibility of a nonadaptive controller using an I-controller that is appropriate for the systems we study. However, the results are applicable to any nonadaptive controller. The unknown and unpredictable behavior of storage systems makes it impossible to develop offline models for them, a fundamental requirement for any type of nonadaptive controller.

The following section describes the design of a controller that dynamically adapts, so that the closed loop is both stable and converges fast independent of workload characteristics and system configuration.

4. DESIGNING AN ADAPTIVE CONTROLLER

We conclude from the previous section, that for a black-box storage system, we need to dynamically adapt the controller as the operating point of the system

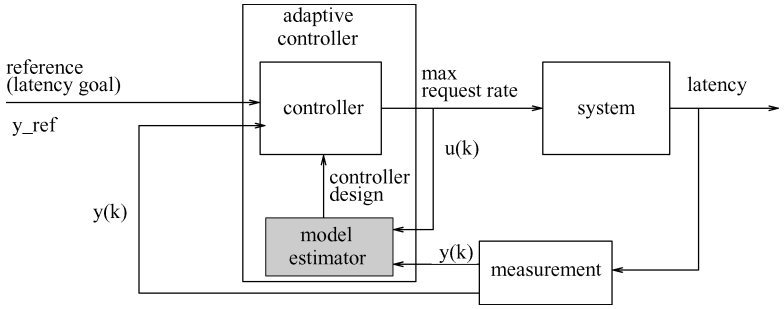


Fig. 6. Adaptive controller for client request throttling.

changes. This is exactly what *adaptive control theory* can be used for. An adaptive controller can be viewed as a controller that performs the system identification process described in the previous section automatically and online. These controllers adapt online to system dynamics in two stages. First, they estimate a system model using an online system identification process. Second, they design online an appropriate controller for the current system model. In practice, online closed-loop design using these two steps is computationally intensive and may result in poorly conditioned loops for some parameter values. Instead, we use a *direct self-tuning regulator* [Åström and Wittenmark 1995] as our adaptive controller. These controllers estimate the system model and controller in a single step, resulting in better adaptivity as well as lower computational complexity. A block diagram of the feedback loop with the adaptive controller is shown in Figure 6.

4.1 Analysis of the Adaptive Closed Loop

The main idea behind direct self-tuning regulators is to estimate a system model that directly captures the controller parameters. In order to construct the control law (the algorithm that decides how to perform actuation), the adaptive controller first estimates a model which is then used to derive the control law. We show how to do this, by starting from the generic model (1) with $N = 1$ and $M = 2$ rewritten as follows:

$$y(k) = s_1 y(k-1) + r_1 u(k-1) + r_2 u(k-2). \tag{7}$$

This is the model of the system from the perspective of the controller. That is, the measured latency, $y(k)$, is a function of the previous actuator settings and latency measurements. We found in Section 3.1 that $y(k)$ sampled at 1-second intervals captures sufficiently the system dynamics. That system identification process also showed that a first-order model accurately describes our target system. However, in order to form a direct self-tuning regulator of a first-order system, we need to start from (7) which is a second-order model. The reason for this will be explained in the following paragraph.

The model parameters of (7) are estimated using a *Recursive Least-Squares* (RLS) estimator [Ljung and Söderström 1987], an online version of the well-known least-squares regression process. It provides the same estimation of regular least squares, but with a lower computational complexity. To derive a

controller from this model, we observe that a controller is a function that returns $u(k)$. By shifting Eq. (7) one step ahead in time and solving for $u(k)$, we get:

$$u(k) = \frac{1}{r_1} y(k+1) - \frac{s_1}{r_1} y(k) - \frac{r_2}{r_1} u(k-1). \tag{8}$$

If this equation is to be used to calculate the actuation setting $u(k)$, then $y(k+1)$ represents the desirable latency to be measured at the next sample point at time $k+1$, that is, it is y_{ref}. Thus, the final control law is:

$$u(k) = \frac{1}{r_1} y_{ref} - \frac{s_1}{r_1} y(k) - \frac{r_2}{r_1} u(k-1) \tag{9}$$

The *stability* of the proposed adaptive controller can be established using a variation of a well-known proof from the literature [Åström and Wittenmark 1995]. That proof applies to a simple direct adaptive control law that uses a gradient estimator. In our case, however, we have a *least-squares estimator*. The proof is adapted to apply to our estimator by ensuring persistent excitation. Persistent excitation just states that there must be enough variability in the actuator settings that RLS can estimate a model from it. Simply put, we need to observe the latency of more than one actuator setting to form a model of it. The rest of the proof steps remain the same. For the proof to be applicable, the closed-loop system must satisfy all of the following properties: (i) the delay d (number of intervals) by which previous controller outputs $u(k)$ affect the system is fixed and known; (ii) the zeroes (roots of the nominator) of the system's transfer function are within the unit circle; (iii) the sign of r_1 is known; and (iv) the upper bound on the order of the system is known. For our system, $d = 1$, the zeroes of the system are at zero, $r_1 > 0$, and we know that our system can be described well by a first-order model. Given that these conditions hold, the proof shows that the following are true: (a) the estimated model parameters are bounded; (b) the normalized model prediction error converges to zero; (c) the actuator setting $u(k)$ and system output $y(k)$ are bounded; (d) the controlled system output $y(k)$ converges to the reference value y_{ref}. Therefore, the closed-loop system with the direct self-tuning regulator is stable, and the system latency converges to the reference latency in steady state. The details of the stability proof can be found in the Appendix

4.2 Adaptive Controller Design

In this section, we describe the operation of the adaptive controller in detail. We discuss a number of heuristics we use to improve the properties of the closed loop, based on knowledge of the specific domain. Using the pseudocode of Figure 7, we go through all the steps of the online controller design process and provide the intuition behind each step.

First, in line 1, the algorithm applies a so-called *conditional update law* [Åström and Wittenmark 1995]. It checks whether there are any requests in the last sample interval to provide a latency measurement. If not, neither the model parameters are modified nor the actuation is computed.

At every sampling period, the algorithm performs an online estimation of the model of the closed-loop system (equation (7)), as described in Section 4.1. That

```
1    if 0 requests
2        exit
3    estimate new model parameters
4    current model = new model + λ * old model
5    if new model very different from old model
6        discard old model
7        u(k) = u_max
8        exit
9    if sign of current model negative or r_1 = 0
10       current model = old model
11   set u(k) according to current model
12   if u(k) < 0
13       u(k) = 0
14   else if u(k) > u_max
15       u(k) = u_max
16   old model = current model
```

Fig. 7. Pseudocode description of adaptive controller.

is, it estimates parameters s_1, r_1 and r_2 using least-squares regression [Åström and Wittenmark 1995] on the measured latencies. As a model derived from just one sample interval is generally not a good one, the algorithm uses a model that is a combination of the previous model and the model calculated from the last interval measurements. The extent that the old model is taken into account is governed by a *forgetting factor* λ, $0 < \lambda \leq 1$.

When the system changes suddenly, the controller needs to adapt faster than what the forgetting factor λ allows. This is handled by the *reset law* of line 5. If any of the new model parameters differ more than 30% from those of the old model, the old model is not taken into account at all. To ensure sufficient excitation so that a good new model can be estimated, $u(k)$ is set to its *maximum value* $u_{\max}$. In the down side, this results in poor workload isolation and differentiation for a few sample intervals. However, it usually pays off, as the new incorrect model produces a few quite different actuation settings. This means that the estimator gets good varied data to quickly estimate a better model and thus overall results in shorter settling times.

There is a possibility that the estimated model predicts a behavior that we know to be impossible in the system. Specifically, it may predict that an increase in throughput results in lower latency or that $r_1 = 0$ (undesirable because r_1 is inverted). This is tested in line 9. As this can never be the case in computer systems, the algorithm discards the new model and uses the one from the previous interval instead. Even if such a wrong model was allowed to be used, the controller would eventually converge to the right model. By including this test, the controller converges faster.

The above three improvements are all targeted at shortening the time it takes for the model to converge. As such, they do not affect the stability proof in the Appendix. RLS and the control loop as a whole is guaranteed to be stable and converge even without these improvements. It will just take longer for it to converge.

Finally, the new operating point $u(k)$ is calculated in line 11 using Eq. (9) with the current model estimates. However, we need to make sure that the controller

does not set $u(k)$ to an impossible value or to a value that does not make sense in the system, that is, either $u(k) < 0$ or $u(k) > u_{\max}$ in our case. This is checked using a so-called *anti-windup law*, in line 12 [Åström and Wittenmark 1995]. In those cases, the value of $u(k)$ is set to 0 and $u_{\max}$ respectively.

The enforcement of such boundary conditions on the values of $u(k)$ is not covered by the stability proof in the Appendix. However, instability due to boundaries enforcement can only occur if the boundary conditions are triggered frequently. This is not the case in our design, for two reasons. First, we never operate the system aiming at a workload throughput close to 0 (i.e., just a few requests per second). Second, we set $u_{\max}$ to a value higher than both the total throughput capacity of the system and the throughput allotment of the highest band. Thus, it is unlikely that the controller sets a $u(k)$ above that value. The boundary conditions are usually triggered only when the model is bad due to a model reset or during the first few seconds after turning the system on. Such brief periods of boundary conditions enforcement do not affect stability. The design of adaptive controllers with boundary conditions (also known as saturation points) and other constraints is an active research area in the control theory community.

Finally, an iteration of the algorithm completes with updating the old model with the new one in line 16.

5. EXPERIMENTAL RESULTS

In this section, we use experimental results to confirm the analytical arguments made in Section 4. We demonstrate the following points about the proposed adaptive controller:

—Its performance is comparable to a nonadaptive controller that has been specifically designed for the current operating range of the system.

—It achieves performance isolation and differentiation among workloads according to performance goals specified as in Section 2.

—It performs well in the face of sudden changes to either system, performance goals, or workloads.

We evaluate the proposed adaptive controller in a Lustre installation. Lustre is a cluster file system for Linux that is designed to achieve high-aggregate throughputs. For the experiments, we use either one or two servers and eight client nodes. All nodes are of the same hardware configuration: 2x PIII CPUs at 1 GHz, 2 GB RAM, and one directly attached Seagate 36GB SCSI Ultra160, 15K rpm hard disk. All nodes are running a RedHat Linux installation using kernel version 2.4.20 with Lustre-specific patches.

As before, we assume a workload per client. All the experiments involve synthetic workloads that can be manipulated as required for the points we need to make. We use IOzone [IOzone 2003] as our workload generator, augmented with an implementation of our throttler. Each client starts an IOzone process that synchronously issues request as fast as the throttler allows it to.

We first compare the performance of our adaptive controller with that of the nonadaptive controller that was designed offline for a specific operating

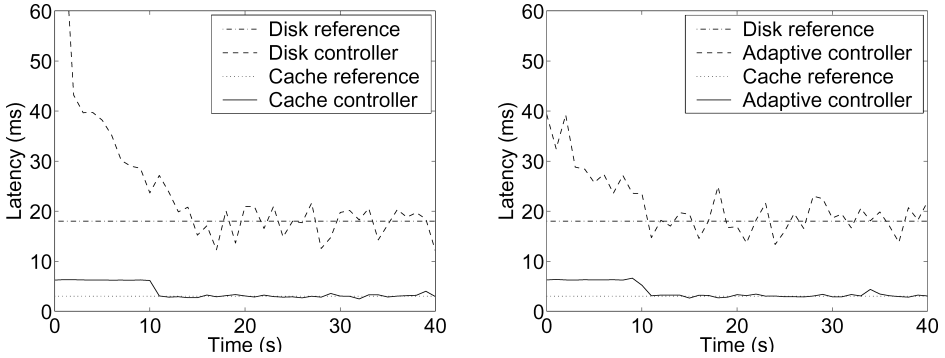


Fig. 8. The performance of the non-adaptive controller on the left and of the adaptive controller on the right. The controllers are enabled at time 10 seconds. Two workloads are considered, a cache-bound one and a disk-bound one, each with its own latency goal.

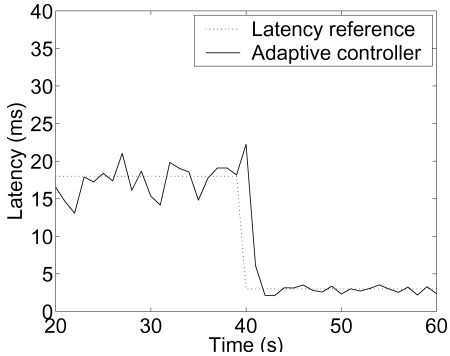


Fig. 9. The performance of the adaptive controller when the workload changes from disk to cache-bound. The latency goal is also changed to demonstrate the adaptability of the model estimation. The workload as well as its latency goal change at time 40 seconds.

range, assuming that the system remains within that operating range. Figure 8 shows that the two controllers are indistinguishable in practice. The adaptive controller has settling times and overshoot comparable to that of the nonadaptive controller (approximately 2–3 seconds). Both controllers result in higher oscillation with the random disk-bound workload, since latencies are more unpredictable in that case. In conclusion, there is no performance penalty due the online estimation of the adaptive controller's parameters.

Figure 9 demonstrates how fast the adaptive controller adapts to sudden system changes. In this case, the workload characteristics change dramatically—the workload turns from an all-in-disk to an all-in-cache data set. To demonstrate adaptability, we also change the latency goal of the workload at the same time, since a more aggressive goal is feasible with the all-in-cache data set. The adaptive controller traces the change and the new performance goal of the workload is met within 3 seconds.

Figure 10 demonstrates performance isolation between two workloads that compete for system throughput. Initially, no throttling is happening and the latency goals of both workloads are violated. The right figure shows the

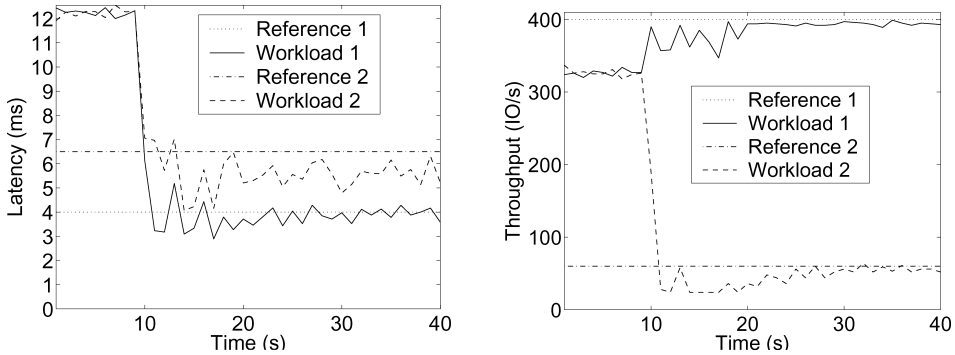


Fig. 10. Latency and throughput isolation when running two cache-bound workloads. The controller is enabled at time 10 seconds.

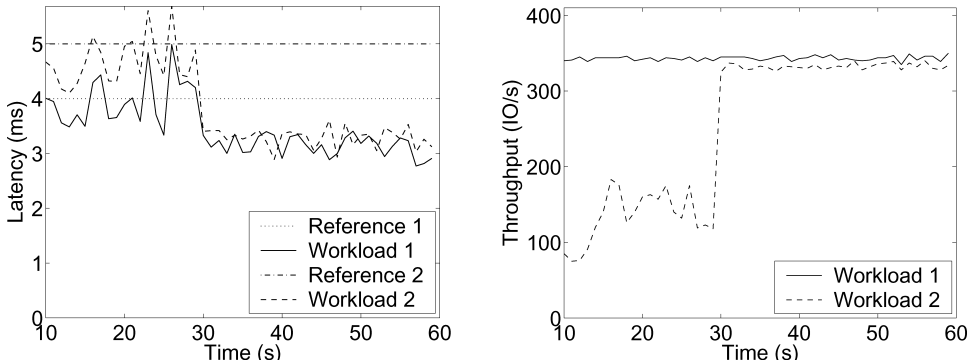


Fig. 11. Latency and throughput differentiation in a dynamic system. Both workloads are cache-bound. At time 30 seconds, the capacity of the system (number of servers) is doubled.

throughput goals of the two workloads—these reference values represent the aggregate throughput goal for each workload as specified by the throughput bands (Section 2), that is achievable with the current system capacity. Before the controller is activated, the throughout goal of workload 2 is exceeded by more than 6x, while the goal of workload 1 is not met. Within approximately 2 seconds from the moment the controller is enabled, the available system throughput is shared between the two workloads according to their specifications. The latency goals of both workloads are also satisfied. In fact, it can be seen, that the aggregate (for both workloads) achievable system throughput with the controller enabled is approximately 150 IO/s less than the aggregate throughput obtained from the uncontrolled system. The reason is that the workloads are throttled so that they meet their latency goals. Higher throughput (even though there is some available capacity in the system) would result in violation of the latency goals, due to queuing delays.

Figure 11 demonstrates differentiation between two workloads, when the capacity of the system is not sufficient to meet the goals of both workloads. It also shows how the controller adapts when the system capacity changes. The performance goals of the two workloads are specified in Table I.

Initially, the data is placed on just one server, which can provide only up to 500 IO/s while satisfying the latency goals of both workloads, which are 4 and 5 ms respectively. According to Table I, the system operates in the beginning of band 2. That is, workload 1 gets all its approximately 350 IO/s due to 50 IO/s from band 0 and 300 IO/s from band 1; workload 2 gets 50 IO/s from band 0 and just some of the IO/s from band 2.

At time 30 seconds, the system's capacity is doubled by adding an additional server. The data is now striped across both servers[6] and both workloads are load-balanced evenly across the two servers. The new system has higher performance, being able to provide close to 700 IO/s while it meets the latency goals of the two workloads. Thus, the system now operates at the end of band 2. The estimated model adapts fast to the change (due to the *reset law* of Figure 7) and the controller changes the throttling performed on workload 2 within 2 seconds from system reconfiguration.

6. CONCLUSION

This article proposes a technique for achieving performance isolation and differentiation among multiple workloads that share the same storage infrastructure, a common problem in consolidated data centers. The proposed solution is based on a distributed adaptive controller that throttles workloads according to their performance goals and their relative importance. The controller considers the storage system as a black box, which makes the solution applicable to a wide range of systems, and it adapts automatically to system and workload dynamics.

The article argues that an adaptive control law is the only appropriate generic way to control a storage system. A nonadaptive controller is not sufficient in our case. We cannot even design a separate nonadaptive controller for each possible operating range of the system, because such operating ranges are not known a priori. Storage systems are large and complex; in general, their performance behavior and workloads cannot be predicted.

In this article, we demonstrate the feasibility and design of an adaptive controller. We do *not* claim that this is the best adaptive controller to be used for black-box storage systems. As a topic for future research, more complex and possibly faster adaptive controllers should be evaluated. However, our arguments about the necessity of adaptive controllers are generally applicable, because of the inherent characteristics of large storage systems and their workloads.

APPENDIX
STABILITY PROOF OF THE ADAPTIVE CONTROLLER

In this Appendix, we prove the stability of our adaptive controller from Section 4. The proof is valid for a changing but bounded y_{ref}, thus we use the notation $y_{ref}(k)$ in the proof. The model used is also more general as it can be of any order. The proof holds for first-order models as well as any other bounded order model.

[6]The size of the data sets for this experiment is small, just a few MB. Thus, the migration of the data to the new configuration is essentially instantaneous—happens within a few ms.

Consider the following discrete-time linear system,

$$w(k) = \phi^T(k-d)\theta_0 = \phi^T(k-1)\theta_0, \tag{10}$$

where

$$\phi(k-1) = [w(k-1)\cdots w(k-N)\ \ u(k-1)\cdots u(k-M)]^T, \tag{11}$$

and

$$\theta_0 = [s_1 \cdots s_N\ r_1 \cdots r_M]^T. \tag{12}$$

LEMMA 1 (RECURSIVE LEAST-SQUARES (RLS) ESTIMATOR PROPERTIES). [GRAHAM AND SANG 1984]. *Let the RLS estimator*

$$\hat{\theta}(k) = \hat{\theta}(k-1) + \frac{P(k-2)\phi(k-1)}{1+\phi(k-1)^T P(k-2)\phi(k-1)} e(k),\ k \geq 1 \tag{13}$$

$$e(k) = w(k) - \phi(k-1)^T \hat{\theta}(k-1) \tag{14}$$

$$P(k-1) = P(k-2) - \frac{P(k-2)\phi(k-1)\phi(k-1)^T P(k-2)}{1+\phi(k-1)^T P(k-2)\phi(k-1)} \tag{15}$$

with $\hat{\theta}(0)$ given and $P(-1) = P(-1)^T > 0$, be applied to data generated by (10)*. It then follows that*

(i) $||\hat{\theta}(k)-\theta_0||^2 \leq \kappa_1 ||\hat{\theta}(0)-\theta_0||^2$, $k \geq 1$, κ_1 = *condition number of* $P(-1)^{-1}$.

(ii) $\lim_{k->\infty} \frac{e(k)}{\sqrt{1+\kappa_2\phi(k-1)^T\phi(k-1))}} = 0$, κ_2 = *maximum eigenvalue of* $P(-1)$.

(iii) $\lim_{k->\infty} ||\hat{\theta}(k) - \hat{\theta}(k-h)|| = 0$, *for any finite h.*

The above lemma shows that (i) parameter estimates from the RLS converges, (ii) the errors in the estimates are bounded, and (iii) the normalized prediction error converges to zero.

LEMMA 2 (KEY TECHNICAL LEMMA [ÅSTRÖM AND WITTENMARK 1995]). *Let* $\{s_k\}$ *be a sequence of real numbers and let* $\{\sigma_k\}$ *be a sequence of vectors such that*

(i) $||\sigma_k|| \leq c_1 + c_2 \max_{0\leq h\leq k} |s_h|$, $c_1 \geq 0$, $c_2 > 0$.

(ii) $\lim_{k->\infty} \frac{s_k^2}{\alpha_1+\alpha_2\sigma_k^T\sigma_k} = 0$, $\alpha_1 > 0$, $\alpha_2 > 0$.

Then $||\sigma_k||$ *is bounded, and* $\lim_{k->\infty} s_k = 0$.

In the direct self-tuning adaptive controller, we use the following control law,

$$\phi(k)^T\hat{\theta}(k) = y_{ref}(k+1), \tag{16}$$

where $y_{ref}(k)$ is the reference value for $w(k)$.

Next we prove a theorem that establishes the stability of the closed-loop system using such a controller. The proof is adapted from the one in Åström and Wittenmark [1995] for an adaptive control law that uses a simple projection algorithm.

THEOREM 1. *Consider a system described by* (10)*. Let the system be controlled with the adaptive control algorithm given by* (16)*, where the estimator is given in Lemma* 1*. Let the reference signal y_{ref} be bounded. Assume that*

(*A*1) *The time delay d from* (10) *is fixed.*
(*A*2) *Upper bounds on the order of the system* (*N,M*) *are known.*
(*A*3) *The zeroes* (*roots of the nominator*) *of the system's transfer function are within the unit circle.*
(*A*4) *The sign of* r_1 *is known.*

Then

(i) *The sequences* $\{u(k)\}$ *and* $\{w(k)\}$ *are bounded.*
(ii) $\lim_{k->\infty} |w(k) - y_{ref}(k)| = 0$.

PROOF. Define the tracking error $\epsilon(k)$ as

$$\epsilon(k) = w(k) - y_{ref}(k). \tag{17}$$

From (16) and (14), we have

$$\epsilon(k) = w(k) - \phi(k-1)^T \hat{\theta}(k-1) = e(k). \tag{18}$$

Hence, the tracking error equals to the prediction error. Then, based on Lemma 1, we have

$$\lim_{k->\infty} \frac{\epsilon(k)^2}{1 + \kappa_2 \phi(k-1)^T \phi(k-1)} = 0.$$

We have established condition (ii) of Lemma 2 with $s_k = \epsilon(k)$, $\sigma_k = \phi(k-1)$, $\alpha_1 = 1$, and $\alpha_2 = \kappa_2$. To establish condition (i), we note that

$$w(k) = \epsilon(k) + y_{ref}(k).$$

Since $y_{ref}(k)$ is bounded, it follows that

$$|w(k)| \le a_1 + a_2 \max_{0 \le h \le k} |\epsilon(k)|, \quad a_1 \ge 0,\ a_2 > 0.$$

Moreover, since the inverse transfer function of the system is stable due to assumption (A3), it follows that

$$|u(k-1)| \le b_1 + b_2 \max_{0 \le h \le k} |\epsilon(k)|, \quad b_1 \ge 0,\ b_2 > 0.$$

Hence, there exist $c_1 \ge 0$ and $c_2 > 0$ such that

$$|\phi(k-1)| \le c_1 + c_2 \max_{0 \le h \le k} |\epsilon(k)|.$$

Applying Lemma 2, we have, $\phi(k)$ is bounded, and $\lim_{k->\infty} |\epsilon(k)| = 0$. Hence, the adaptive controller used in this article is stable. □

ACKNOWLEDGMENTS

The authors would like to thank Terril Hurst, Mikael Johansson and Kim Keeton for their help.

REFERENCES

ABDELZAHER, T., SHIN, K. G., AND BHATTI, N. 2002. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Paral. Distrib. Syst. 13*, 1, 80–96.

ABDELZAHER, T., SHIN, K. G., AND BHATTI, N. 2003. User-level QoS-adaptive resource management in server end-systems. *IEEE Trans. Comput. 52*, 5, 678–685.

ABDELZAHER, T. F. AND BHATTI, N. 1999. Web content adaptation to improve server overload behavior. In *Proceedings of the International World Wide Web Conference (WWW)* (Toronto, Ont., Canada). 1563–1577.

ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. 2002. Hippodrome: Running circles around storage administration. In *Proceedings of the International Conference on File and Storage Technologies (FAST)* (Monterey, CA). 175–188.

ÅSTRÖM, K. J. AND WITTENMARK, B. 1995. *Adaptive Control*, 2nd ed. Electrical Engineering: Control Engineering. Addison-Wesley Publishing Company. ISBN 0-201-55866-1.

CALLAGHAN, B., PAWLOWSKI, B., AND STAUBACH, P. 1995. RFC1813: NFS version 3 protocol specification. http://www.faqs.org/rfcs/rfc1813.html.

CHAMBLISS, D., ALVAREZ, G., PANDEY, P., JADAV, D., XU, J., MENON, R., AND LEE, T. 2003. Performance virtulization for large-scale storage systems. In *Proceedings of the Symposium on Reliable Distributed Systems (SRDS)* (Florence, Italy). 109–118.

DIAO, Y., HELLERSTEIN, J., AND PAREKH, S. 2002a. MIMO control of an Apache web server: Modeling and controller design. In *Proceedings of the American Control Conference (ACC)* (Anchorage, AK). 4922–4927.

DIAO, Y., HELLERSTEIN, J., AND PAREKH, S. 2002b. Optimizing quality of service using fuzzy control. In *Proceedings of the International Workshop on Distributed Systems Operations and Management (DSOM)* (Montreal, Que., Canada). 42–53.

DIAO, Y., HELLERSTEIN, J., AND PAREKH, S. 2002c. Using fuzzy control to maximize profits in service level management. *IBM Syst. J. 41*, 3, 403–420.

DIAO, Y., HELLERSTEIN, J., PAREKH, S., AND BIGUS, J. 2003a. Managing web server performance with AutoTune agents. *IBM Syst. J. 42*, 1, 136–149.

DIAO, Y., LUI, X., FROEHLICH, S., HELLERSTEIN, J., PAREKH, S., AND SHA, L. 2003b. On-line response time optimization of an apache web server. In *Proceedings of the International Workshop on Quality of Service (IWQoS)* (Monterey, CA). 461–478.

FRANKLIN, G. F., POWELL, J. D., AND WORKMAN, M. 1998. *Digital Control of Dynamic Systems*, 3rd ed. Addison-Wesley. ISBN 0-201-82054-4.

GOYAL, P., JADAV, D., MODHA, D., AND TEWARI, R. 2003. CacheCOW: QoS for storage system caches. In *Proceedings of the International Workshop on Quality of Service (IWQoS)* (Monterey, CA). 498–516.

GRAHAM, G. AND SANG, S. K. 1984. *Adaptive Filtering: Prediction and Control*. Prentice Hall, Englewood Cliffs, NJ. ISBN 0-130-04069-X.

HELLERSTEIN, J., DIAO, Y., PAREKH, S., AND TILBURY, D. 2004. *Feedback Control of Computing Systems*. Wiley-IEEE Press, New York. ISBN 0-471266-37-X.

IOZONE 2003. *IOzone File-System Benchmark*. www.iozone.org.

KO, B.-J., LEE, K.-W., AMIRI, K., AND CALO, S. 2003. Scalable service differentiation in a shared storage cache. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)* (Providence, RI). 184–193.

LEE, H. D., NAM, Y. J., AND PARK, C. 2004. Regulating I/O performance of shared storage with a control theoretical approach. In *Proceedings of the NASA / IEEE Conference on Mass Storage Systems and Technologies (MSST)* (College Park, MD). IEEE Computer Society Press, Los Alamitos, CA.

LI, B. AND NAHRSTEDT, K. 1998. A control theoretical model for quality of service adaptations. In *Proceedings of the International Workshop on Quality of Service (IWQoS)* (Napa, CA). 145–153.

LJUNG, L. 1998. *System Identification: Theory for the User*, 2nd ed. Prentice-Hall, Englewood, Cliffs, NJ. ISBN 0-136566-95-2.

LJUNG, L. AND SÖDERSTRÖM, T. 1987. *Theory and Practise of Recursive Identification*. MIT Press, Cambridge, MA. ISBN 0-262620-58-8.

LU, C., ABDELZAHER, T., STANKOVIC, J., AND SON, S. 2001. A feedback control approach for guaranteeing relative delays in web servers. In *Proceedings of the IEEE Real Time Technology and Applications Symposium (RTAS)* (Taipei, Taiwan), IEEE Computer Society Press, Los Alamitos, CA. 51–62.

LU, Y., ABDELZAHER, T., LU, C., AND TAO, G. 2002. An adaptive control framework for QoS guarantees and its application to differentiated caching services. In *Proceedings of the International Workshop on Quality of Service (IWQoS)* (Miami Beach, FL). 23–32.

LU, Y., SAXENA, A., AND ABDELZAHER, T. 2001. Differentiated caching services; a control-theoretical approach. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)* (Phoenix, AZ). 615–622.

LUMB, C., MERCHANT, A., AND ALVAREZ, G. 2003. Façade: Virtual storage devices with performance guarantees. In *Proceedings of the International Conference on File and Storage Technologies (FAST)* (San Francisco, CA). 131–144.

LUSTRE. 2005. *Lustre Cluster File-System*. www.lustre.org.

PAREKH, S., HELLERSTEIN, J., JAYRAM, T., GANDHI, N., TILBURY, D., AND BIGUS, J. 2002. Using control theory to achieve service level objectives in performance management. *J. Real-Time Syst. 23*, 1–2 (July-Sept.), 127–141.

ROBERTSSON, A., WITTENMARK, B., AND KIHL, M. 2003. Analysis and design of admission control in web-server systems. In *Proceedings of the American Control Conference (ACC)* (Denver, CO). 254–259.

ROBERTSSON, A., WITTENMARK, B., KIHL, M., AND ANDERSSON, M. 2004. Admission control for web server systems-design and experimental evaluation. In *Proceedings of the IEEE Conference on Decision and Control (CDC)* (Paradise Island, Bahamas). IEEE Computer Society Press, Los Alamitos, CA, 531–536.

SAITO, Y., FRØLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. 2004. Fab: building reliable enterprise storage systems on the cheap. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Boston, MA). 48–58.

STANKOVIC, J., HE, T., ABDELZAHER, T., MARLEY, M., TAO, G., AND SO, S. 2001. Feedback control scheduling in distributed systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)* (London, UK). 59–72.

SUNDARAM, V. AND SHENOY, P. 2003. A practical learning-based approach for dynamic storage bandwidth allocation. In *Proceedings of the International Workshop on Quality of Service (IWQoS)* (Monterey, CA) 479–497.

WELSH, M. AND CULLER, D. 2003. Adaptive overload control for busy internet servers. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)* (Seattle, WA). 43–56.

Received April 2005; revised July 2005; accepted August 2005