Triage: Performance Isolation and Differentiation for Storage Systems

Magnus Karlsson, Christos Karamanolis and Xiaoyun Zhu
HP Labs, Palo Alto, CA
{karlsson,christos,xiaoyun}@hpl.hp.com

Abstract

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 depend on 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 impossible 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 on-line 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.

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 of multiple "customers". It is important to ensure that customers receive the resources and performance they are entitled to. More specifically, the performance of workloads must be isolated from the activities of other workloads that share the same infrastructure. Further, 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 [4]. 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. In fact, it may be impossible to do that, since the worst-case scenario 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 paper, we focus on storage system *throughput* as the key resource that is shared by the workloads. Throughput reflects the capacities of different 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 should depend on the behavior of system components, their configuration, as well as workload dynamics. 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 models of such systems off-line to make performance predictions, as has been proposed in the literature [1, 9, 19, 22, 29].

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 results in higher request latencies and that the order of the system model is known. We solely depend on on-line performance monitoring from outside the system to infer system models and perform workload arbitration accordingly.

More specifically, we use an on-line 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 according to a specified resource sharing policy.

To compensate for the lack of known models and to obtain guaranteed stability and system performance, we use a control-theoretic approach for the design of the feedback loop. In particular, we propose using a *direct self-tuning adaptive controller*, the parameters of which adapt to system and workload dynamics on-line without prior tuning.

Existing systems that apply control theory to computer systems (LotusNotes [28], Apache [1, 2, 3, 9, 22, 29], Squid [25], middleware [21], file server [20], data migration [8]) use non-adaptive controllers that are designed off-line. Other systems [7, 10, 11, 12, 13, 16, 26, 31, 32] that do not use control theory still require prior tuning (because they are non-adaptive) and/or modifications to the target system. Storage systems demonstrate different non-linear behavior depending on system configuration and the workloads. For example, a workload that retrieves data from an internal cache has very different behavior from one that gets data from disk. We show in this paper that, in the general case, it is not possible to design a well behaved non-adaptive linear controller with parameters that are applicable to all different operating ranges of a black-box storage system, because of the large variability in the operating ranges. The use of such a controller would result in long settling times or even instability of the system. This precludes the use of any of the prior-art mentioned.

There is one case in the literature where an adaptive controller was used to control application performance [24]. This solution requires modifications of the controlled system, a web cache in this case. In our case, *no modifications* are required for the target system. In our current prototype, *Triage*, throttling is performed on the clients. It can be implemented either by modifying the storage access protocol or transparently in a virtualization layer (e.g., a logical volume manager or a virtual machine monitor). In addition, Triage does not require any centralized point of control. The controller is implemented in a *distributed* fashion with a module on each client, something we have only seen in two papers [23, 30].

2 Specifying performance objectives

As discussed in section 1, this paper proposes an on-line 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 impossible 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 target* that should be met for all workload requests. This latency 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.[1]

These are both soft goals. 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. We observe that users do not 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 some progress. Above that, the value customers assign to the required throughput typically declines, but with different rates for various workloads. To capture such varying cost functions for throughput, we specify a number of *bands* for the available system throughput.

Table 1: *Example of two workloads sharing the system according to three throughput bands. 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. Any available throughput beyond band 2 is shared fairly (50-50) between the workloads.*

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

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 breaks. So will W1, if it does not get 50 IO/s. To satisfy the combined throughput requirements of the two workloads, we specify the three *bands* for throughput sharing, as shown in Table 1. 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

[1]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 paper.

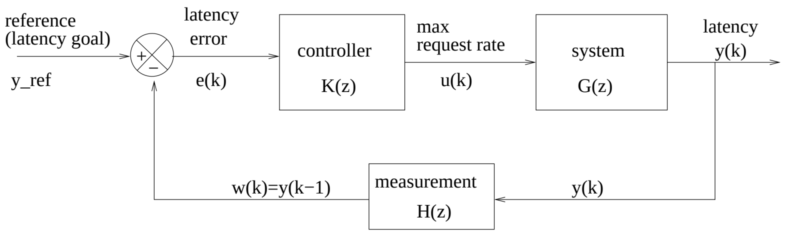


Figure 1: *Feedback loop for client request throttling.*

"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.

In addition, the latency target of each workload should be met in the system. At an instance in time, the system is operating in a band i. As soon as the latency goal of at least one workload with a non-zero 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 non-zero 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. For simplicity, we assume that there is a 1:1 mapping between clients and workloads. Examples of such systems include network file systems [6], cluster file systems [27], or block-based storage [15]. For the discussions in this paper, we use an installation of a cluster file system, Lustre [27], with 8 clients and 1 or 2 servers.

The objective is to design a feedback controller that arbitrates the usage of system throughput by throttling client requests according to the specifications of the throughput bands. Since we cannot instrument the system to either perform throttling or to obtain measurements, we require that the feedback loop depends merely on externally observed metrics of the system's performance, i.e., response latency and throughput.

Figure 1 shows an abstract representation of the feedback loop. In the figure, the output $u(k)$ of the controller is the desired *operating point* of the system. $y(k)$ is the latency of the system averaged over some sampling period, the length of which is specified in the system identification process described later. The input to the closed-loop system, y_{ref}, is the reference value for $y(k)$. The input $e(k)$ to the controller is the latency error, i.e., the difference between the measured and the target latency values. Based on the observed latency error, 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. Enforcing this maximum throughput requires that a throttling module intercepts requests somewhere on the data path—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 practical 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).

In practice, there is a feedback loop for each client/workload in the system. As shown in Figure 2, there are a controller and a throttler module on each client. The reference input to the controller is the latency goal for this 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 paper). The throttler imposes a maximum request rate for outgoing requests from the corresponding client.

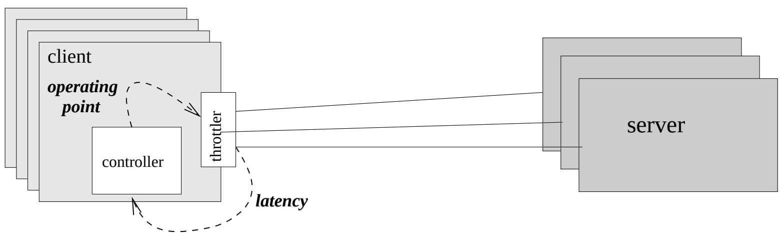


Figure 2: *A controlled Lustre instantiation.*

In this section, we describe the design of a feedback control loop using a non-adaptive controller. We do this for four reasons. First, we show that non-adaptive 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 off-line system identification technique forms the basis for the on-line estimation technique used for the adaptive controller. Four, we derive the order of the system model, which is also used for the adaptive closed loop.

3.1 System identification and validation

The first step toward designing a non-adaptive controller is to develop a model of the target system. Since we consider the system as a black box, we use statistical methods to obtain the model. That is, we excite the system with white noise input signal ($u(k)$), since white noise consists of signals that cover the entire spectrum of potential input frequencies. We implemented this 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 (8 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.

The response latency for every request sent to the system is measured on each client. The measured latencies are averaged over every sampling interval. System identification involves fitting these measured average values to the following discrete-time linear model [14], by using least-squares regression (LSR)[2] [5].

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

In this model, $y(k)$ is the latency of the requests at time k and $u(k)$ is the operating point set at time k. The number N is the order of the system, which captures the extent of correlation between the system's current and past states.

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 is usually sufficient to capture the dynamics of the system, as there are few requests that affect the system in 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 (or longer) than the sampling period, then higher order systems are required. Intuitively, a long sampling period may result in slow reaction and thus insufficient actuation by the controller. On the other hand, a short sampling period

[2] 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.

Table 2: *R^2 fit of a first-order model and residual correlation coefficient as a function of sample interval. Two workloads: (i) all accesses in the cache; (ii) all accesses on random locations on disk.*

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

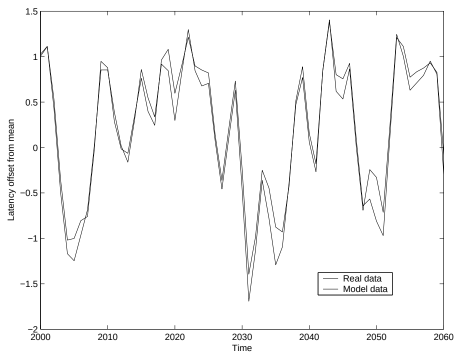
may result in considerable measurement noise and model over-fitting [14], which in turn leads to oscillations in the controlled system.

Table 2 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* [14] 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* [14] 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.

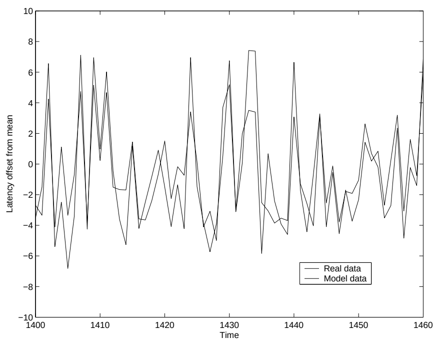
We observe in Table 2, that a first-order model extracts most of the information from the data. The two exceptions are the 300 and 100 ms intervals for the in-cache case. We have tried higher-order models for these cases, but they resulted in less than 0.05 improvement to R^2 fits—they are still a lot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

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,



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



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

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

$y(k)$ 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 $k-1$.[3]

3.2 Non-adaptive 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 that for a first-order system like ours, a simple *integral* (I) controller suffices to control the system [14]. 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 at 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 [14]. 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.

[3]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 100.

However, to choose a value for K_I that leads to a stable system with low settling times and low overshoot, we need to analyze the closed-loop transfer function using its Z-transform [14]. To do this, we first need the Z-transform of the controller, $K(z)$, which can be derived from (2) as follows:

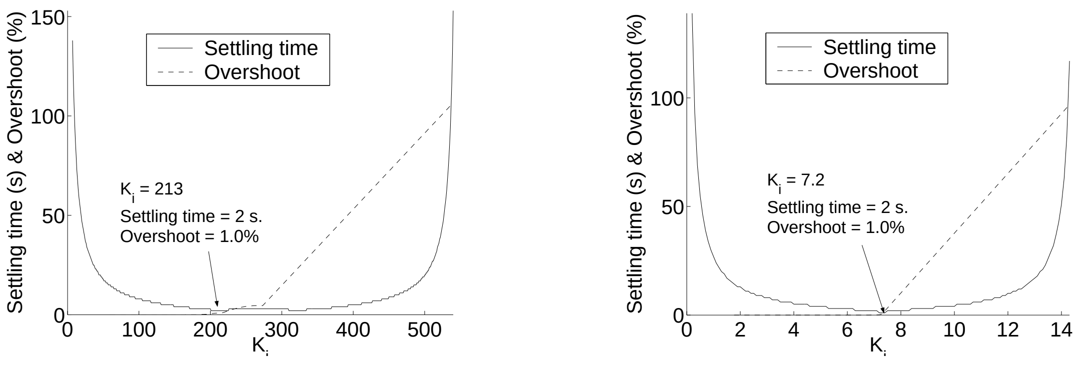
$$U(z) = U(z)z^{-1} + K_I E(z) \Rightarrow K(z) = \frac{U(z)}{E(z)} = \frac{zK_I}{z-1} \tag{3}$$

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 [14]. It is as follows:

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

In this equation, $K(z)$ is given by (3) and $G(z) = G_c(z) = \frac{0.003827z}{z-0.04554}$ or $G(z) = G_d(z) = \frac{0.1377z}{z-0.001109}$ respectively, for each of the two system models of Figure 3. $H(z) = z^{-1}$ representing a worst-case delay of one interval for the real system latency to be observed by the controller due to the averaging. Inserting these values into (4), we obtain two versions of the system's transfer function, one for each of the system models. 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. 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. 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.



(i) (ii)

Figure 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 (%).*

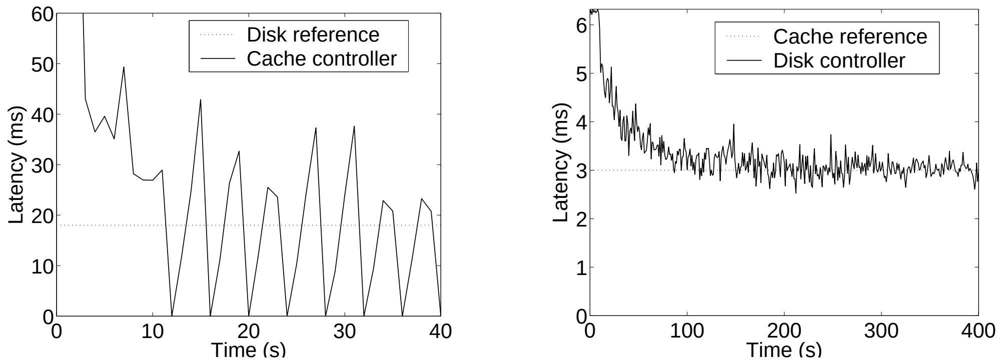


Figure 5: *The performance of the non-adaptive controller. A controller designed for the in-cache model is applied to a mostly on-disk workloads (left). A controller for the on-disk model is applied to a a mostly in-cache workload (right). A latency of 0 means that there were no requests during that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

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

The following section describes the design of a controller that dynamically adapts to be both stable and 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 range of the system changes. This is exactly what *adaptive control theory* can be used for. Most adaptive controllers estimate a model *on-line* using least-squares regression. This is the same regression technique that is used in Section 3. However, the model is now estimated at every sampling period and it is used for on-line controller design. Again, the resulting control loop must meet certain properties, namely stability, fast settling times and low overshoot. In practice, on-line closed-loop design using these two steps (model estimation and controller design) may be time consuming and may result in poorly conditioned loops for some parameter values. Instead, we use a *direct self-tuning regulator* [5] as our adaptive controller. These controllers estimate the control loop parameters (including the controller parameters) in a single step, resulting in better adaptivity as well as simpler mathematical formulations. On the down side, it is harder to provide intuitive explanations of their behavior. A block diagram of the feedback loop with the adaptive controller is shown in Figure 6(ii).

4.1 Analysis of the adaptive closed loop

From the analysis of the non-adaptive controller, we know that the system can be captured by a first-order model and that a simple I-controller works well (if we could adapt K_I). Thus, we use an I-controller for the adaptive case too. The main idea behind a direct self-tuning regulator is to estimate a system model that directly captures the controller parameters. In order to construct an integral control law, the adaptive controller first needs to estimate a model for the system that can be turned into an I-controller. We will show how to do this, by starting from the following generic model.

$$w(k) = s_1 w(k-1) + r_1 u(k-1) + r_2 u(k-2) \tag{5}$$

This is the model of the system from the perspective of the controller. That is, the measured latency, $w(k)$, is a function of the previous actuator settings and measurements. The model parameters of (5) are estimated using a *Recursive Least-Squares* (RLS) estimator, an on-line version of the LSR process. To turn this model into a controller, we observe that a controller is a function that returns $u(k)$. If we shift equation (5) one step ahead in time and solve for $u(k)$, we get:

$$u(k) = \frac{1}{r_1} w(k+1) - \frac{s_1}{r_1} w(k) - \frac{r_2}{r_1} u(k-1) \tag{6}$$

```
if < 6 requests
    exit
estimate new model parameters
current model = new model + $\lambda$ * old model
if new model very different from old model
    discard old model
    $u(k) = u_{max}$
    exit
if sign of current model negative or $r_1 = 0$
    current model = old model
set $u^*(k)$ according to current model
if $u^*(k) < 0$
    $u(k) = 0$
else if $u^*(k) > u_{max}$
    $u(k) = u_{max}$
else
    $u(k) = u^*(k)$
old model = current model
```

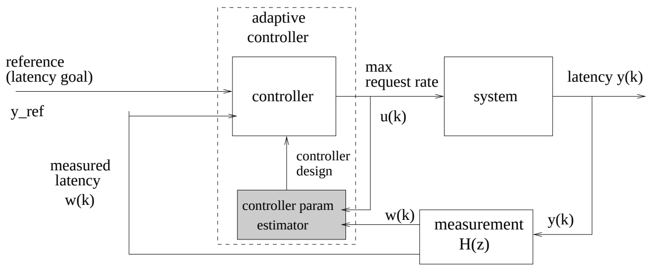


Figure 6: *The design of the adaptive controller. (i) Pseudo-code description of adaptation algorithm. (ii) Block diagram of feedback loop with adaptive controller.*

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

$$u(k) = \frac{1}{r_1} y_{ref} - \frac{s_1}{r_1} w(k) - \frac{r_2}{r_1} u(k-1) \tag{7}$$

The *stability* of the proposed adaptive controller can be established using a variation of a well-known proof from the literature [5]. 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 so that the estimated covariance matrix stays bounded. The rest of the proof steps remain the same. For the proof to be applicable, the closed-loop system must satisfy all the following properties: (i) the delay d (number of intervals) by which previous intputs $u(k)$ affect the system; (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 from Section 3.1 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 $w(k)$ are bounded; (d) the controlled system output $w(k)$ converges to the reference value y_{ref}. Therefore, our closed-loop system with the direct self-tuning regulator is shown to be 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 pseudo-code of Figure 6 (i), we go through all the steps of the on-line controller design process and provide the intuition behind each step.

First, in line 1, the algorithm applies a so-called *conditional update law* [5]. It checks whether there are enough requests in the last sample interval for it to be statistically significant—at least 6 requests are required. Otherwise, neither the model parameters are modified nor the actuation is changed. To avoid potential system deadlock when all controllers decide not to do any changes (e.g., the system becomes suddenly extremely loaded and slow because of a component failure), one random controller in the system does not execute this if-statement. This ensures that one control loop is always executed and affects the system.

At every sample period, the algorithm performs an on-line estimation of the model of the closed-loop system (equation (5)), as described in Section 4.1. That is, it estimates parameters s_1, r_1 and r_2 using least-squares regression [5] 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* $\lambda, 0 < \lambda \leq 1$.

When the system changes suddenly, the controller needs to adapt faster than what the forgetting factor λ allows. This case 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 for the system, $u(k)$ 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 pays off, as high excitation means a better model and thus the loop settles faster.

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 $r_1 = 0$. 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 of 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, the controller converges faster.

Finally, the new operating point $u(k)$ is calculated in line 11 using equation (7) with the current model estimates. However, we need to make sure that the controller does not set $u(k)$ to an impossible value, either $u(k) < 0$ or $u(k) > u_{max}$. This is checked using an *anti-windup law*, in line 12. In those cases, the value of $u(k)$ is set to 0 and u_{max} respectively. Not having this anti-windup safeguard might make the controller unstable if it spent several sample periods with values below 0 or above u_{max} [5]. An iteration of the algorithm completes with updating the old model with the new one in line 18.

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:

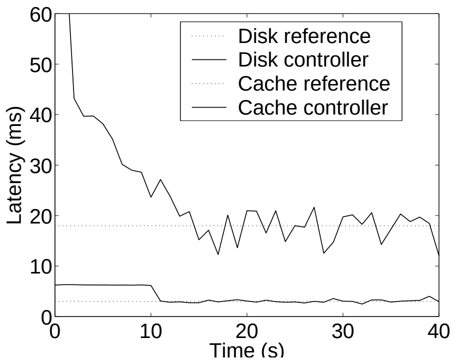
- Its performance is comparable to a non-adaptive controller that has been specifically designed for the current operating range of the system.
- It achieves performance isolation and differentiation among workloads according to performance goals as specified in Section 2.
- It performs well in the face 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 with 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 [18] 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 an non-adaptive controller that could have been designed off-line for a specific operating range; assuming that the system remains within that operating range. Figure 7 shows that the two controllers are indistinguishable in practice. The adaptive controller has settling times and overshoot comparable to that of the non-adaptive controller (approximately 2-3 s). 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 on-line estimation of the adaptive controller's parameters.

Figure 8 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



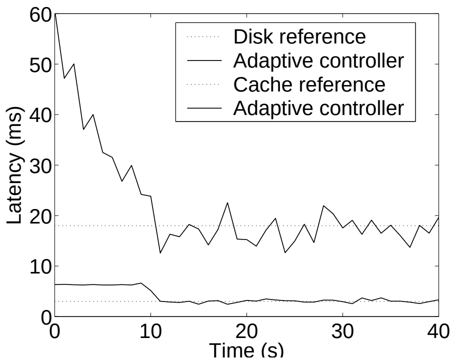


Figure 7: *The performance of the non-adaptive controller on the left and of the adaptive controller on the right. The controllers are enabled at time 10 s. Two workloads are considered, a cache-bound one and a disk-bound one, each with its own latency goal.*

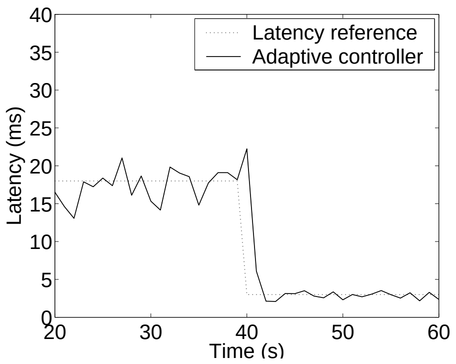
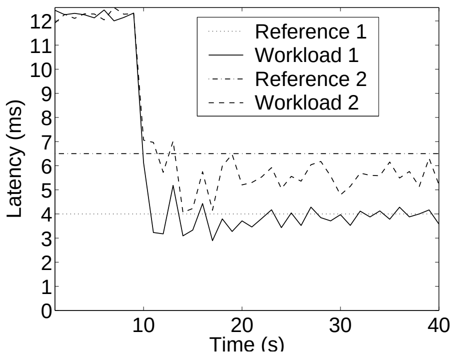


Figure 8: *The performance of the adaptive controller when the workload changes from disk to cache-bound. The latency goal is also changed to demonstrate the adaptability of the model estimation. The workload as well as its latency goal change at time 40 s.*



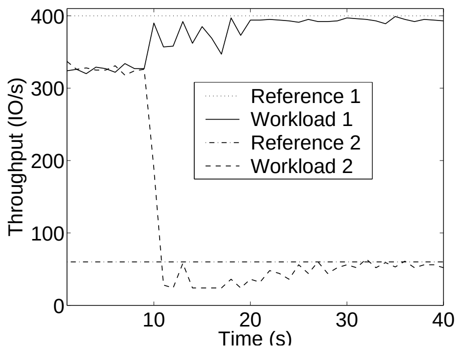


Figure 9: *Latency and throughput isolation when running two workloads. Both workloads are cache-bound. The controller is enabled at time 10 s.*

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 s.

Figure 9 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

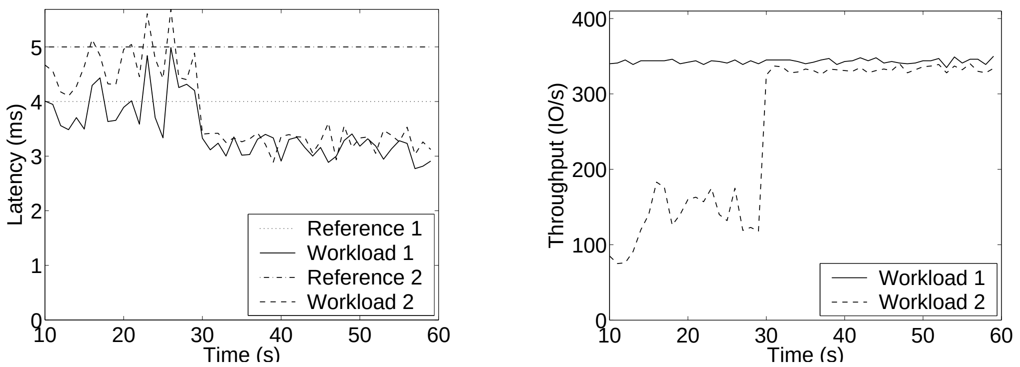


Figure 10: *Latency and throughput differentiation in a dynamic system. Both workloads are cache-bound. At time 30 s, the capacity of the system (number of servers) is doubled.*

figure shows the throughput goals of the two workloads—these reference values represent the aggregate throughput goal for each workload as specified by a number of throughput bands (Section 2). 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 s 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 10 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 1.

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 1, 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 s, the system's capacity is doubled by adding an additional server. The data is now striped across both servers[4] 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

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

change (due to the *reset law* of Figure 6) and the controller changes the throttling performed on workload 2 within 2 s from the system reconfiguration.

6 Conclusion

This paper 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 paper argues that a traditional non-adaptive controller is not sufficient in our case. We cannot even design a separate non-adaptive 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 the dynamics of their workloads cannot be predicted.

Thus, an adaptive control law is the only possible generic way to control a storage system. In this paper, we make our arguments, both for the infeasibility of non-adaptive controllers and the design of an adaptive one, on the basis of a simple control law. We look into *integral controllers*, that are simple to analyze, but still work well for first-order systems, such as the storage systems we have studied. Having said this, we do *not* claim that an I-controller is necessarily the best control law for black-box storage systems. As a topic for future research, more complex and generally faster control laws, such as PID and MIMO [14], should be evaluated. However, our arguments about the infeasibility of non-adaptive controllers are generally applicable because of the inherent characteristics of large storage systems and their workloads.

We are currently implementing our controller in a storage virtualization platform, for seamless integration with multiple OSs and different storage access protocols. We are also studying the mapping of high-level business and application objectives on system performance goals, like the ones used in this paper. Last but not least, we are looking into how to integrate our approach with provisioning tools.

References

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

[2] T. Abdelzaher, K. G. Shin, and N. Bhatti. User-level QoS-adaptive resource management in server end-systems. *IEEE Transactions on Computers*, 52(5), 2003.

[3] T. F. Abdelzaher and N. Bhatti. Web content adaptation to improve server overload behavior. In *International World Wide Web Conference (WWW)*, Toronto, Canada, May 1999.

[4] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *International Conference on File and Storage Technologies (FAST)*, pages 175–188, Monterey, CA, January 2002.

[5] K. J. Åström and B. Wittenmark. *Adaptive Control*. Series in Electrical Engineering: Control Engineering. Addison-Wesley Publishing Company, 2 edition, 1995. ISBN 0-201-55866-1.

[6] B. Callaghan, B. Pawlowski, and P. Staubach. RFC1813: NFS version 3 protocol specification. http://www.faqs-.org/rfcs/rfc1813.html, June 1995.

[7] D. Chambliss, G. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. Lee. Performance virtulization for large-scale storage systems. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 109–118, Florence, Italy, October 2003.

[8] L. Chenyang, A. Guillermo, and W. John. Aqueduct: online data migration with performance guarantees. In *International Conference on File and Storage Technologies (FAST)*, pages 219–230, Monterey, CA, January 2002.

[9] Y. Diao, J. Hellerstein, and S. Parekh. MIMO control of an Apache web server: Modeling and controller design. In *American Control Conference (ACC)*, pages 4922–4927, Anchorage, AK, May 2002.

[10] Y. Diao, J. Hellerstein, and S. Parekh. Optimizing quality of service using fuzzy control. In *International Workshop on Distributed Systems Operations and Management (DSOM)*, pages 42–53, Montreal, Canada, October 2002.

[11] Y. Diao, J. Hellerstein, and S. Parekh. Using fuzzy control to maximize profits in service level management. *IBM Systems Journal*, 41(3):403–420, 2002.

[12] Y. Diao, J. Hellerstein, S. Parekh, and J. Bigus. Managing web server performance with AutoTune agents. *IBM Systems Journal*, 42(1):136–149, 2003.

[13] Y. Diao, X. Lui, S. Froehlich, J. Hellerstein, S. Parekh, and L. Sha. On-line response time optimization of an apache web server. In *International Workshop on Quality of Service (IWQoS)*, pages 461–478, Monterey, CA, June 2003.

[14] G. F. Franklin, J. D. Powell, and M. Workman. *Digital Control of Dynamic Systems*. Addison-Wesley Publishing Company, 3 edition, 1998. ISBN 0-201-82054-4.

[15] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: enterprise storage systems on a shoestring. In *Workshop on Hot Topics in Operating Systems (HotOS)*, pages 169–174, Lihue, HI, May 2003.

[16] P. Goyal, D. Jadav, D. Modha, and R. Tewari. CacheCOW: QoS for storage system caches. In *International Workshop on Quality of Service (IWQoS)*, pages 498–516, Monterey, CA, June 2003.

[17] G. Graham and S. K. Sang. *Adaptive Filtering: Prediction and Control*. Prentice Hall, March 1984. ISBN 0-130-04069-X.

[18] *IOzone File-System Benchmark*. www.iozone.org.

[19] B.-J. Ko, K.-W. Lee, K. Amiri, and S. Calo. Scalable service differentiation in a shared storage cache. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 184–193, Providence, RI, May 2003.

[20] H. D. Lee, Y. J. Nam, and C. Park. Regulating I/O performance of shared storage with a control theoretical approach. In *NASA/IEEE Conference on Mass Storage Systems and Technologies (MSST)*, College Park, ML, April 2004.

[21] B. Li and K. Nahrstedt. A control theoretical model for quality of service adaptations. In *International Workshop on Quality of Service (IWQoS)*, pages 145–153, Napa, CA, May 1998.

[22] C. Lu, T. Abdelzaher, J. Stankovic, and S. Son. A feedback control approach for guaranteeing relative delays in web servers. In *IEEE Real Time Technology and Applications Symposium (RTAS)*, pages 51–62, Taipei, Taiwan, June 2001.

[23] C. Lu, X. Wang, and X. Koutsoukos. End-to-end utilization control in distributed real-time systems. In *International Conference on Distributed Computing Systems (ICDCS)*, Tokyo, Japan, March 2004.

[24] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework for QoS guarantees and its application to differentiated caching services. In *International Workshop on Quality of Service (IWQoS)*, pages 23–32, Miami Beach, FL, May 2002.

[25] Y. Lu, A. Saxena, and T. Abdelzaher. Differentiated caching services; a control-theoretical approach. In *International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, AZ, April 2001.

[26] C. Lumb, A. Merchant, and G. Alvarez. Façade: Virtual storage devices with performance guarantees. In *International Conference on File and Storage Technologies (FAST)*, pages 131–144, San Francisco, CA, March 2003.

[27] *Lustre Cluster File-System*. www.lustre.org.

[28] S. Parekh, J. Hellerstein, T. Jayram, N. Gandhi, D. Tilbury, and J. Bigus. Using control theory to achieve service level objectives in performance management. *Journal of Real-Time Systems*, 23(1-2):127–141, July-September 2002.

[29] A. Robertsson, B. Wittenmark, and M. Kihl. Analysis and design of admission control in web-server systems. In *American Control Conference (ACC)*, Denver, CO, June 2003.

[30] J. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, and S. So. Feedback control scheduling in distributed systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 59–72, London, UK, December 2001.

[31] V. Sundaram and P. Shenoy. A practical learning-based approach for dynamic storage bandwidth allocation. In *International Workshop on Quality of Service (IWQoS)*, pages 479–497, Monterey, CA, June 2003.

[32] M. Welsh and D. Culler. Adaptive overload control for busy internet servers. In *USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 43–56, Seattle, WA, March 2003.

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.

Consider the following discrete-time linear system,

$$w(k) = \phi^T(k-d)\theta_0 = \phi^T(k-1)\theta_0 \tag{8}$$

where

$$\phi(k-1) = [w(k-1) \ \ \dots \ \ w(k-N) \ \ u(k-1) \ \ \dots \ \ u(k-M)]^T, \tag{9}$$

and

$$\theta_0 = [s_1 \ \ \dots s_N \ \ r_1 \ \ \dots \ \ r_M]^T. \tag{10}$$

Lemma 1 Recursive Least-Squares (RLS) estimator properties *[17]*

Let the RLS estimator

$$\hat{\theta}(k) = \hat{\theta}(k-1) + \frac{P(k-2)\phi(k-1)}{1+\phi(k-1)^T P(k-2)\phi(k-1)} e(k), \ k \geq 1 \tag{11}$$

$$e(k) = w(k) - \phi(k-1)^T \hat{\theta}(k-1) \tag{12}$$

$$P(k-1) = P(k-2) - \frac{P(k-2)\phi(k-1)\phi(k-1)^T P(k-2)}{1+\phi(k-1)^T P(k-2)\phi(k-1)} \tag{13}$$

with $\hat{\theta}(0)$ given and $P(-1) = P(-1)^T > 0$, be applied to data generated by (8). It then follows that

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

(ii) $\lim_{k->\infty} \frac{e(k)}{\sqrt{1+\kappa_2\phi(k-1)^T\phi(k-1))}} = 0, \quad \kappa_2 =$ *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 [5]*

Let $\{s_k\}$ *be a sequence of real numbers and let* $\{\sigma_k\}$ *be a sequence of vectors such that*

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

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

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,

$$\phi(k)^T\hat{\theta}(k) = y_{ref}(k+1) \tag{14}$$

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 [5] for an adaptive control law that uses a simple projection algorithm.

Theorem 1 *Consider a system described by (8). Let the system be controlled with the adaptive control algorithm given by (14), where the estimator is given in Lemma 1. Let the reference signal* y_{ref} *be bounded. Assume that*

A1 *The time delay d from (8) is fixed.*

A2 *Upper bounds on the order of the system (N,M) are known.*

A3 *The zeroes (roots of the nominator) of the system's transfer function are within the unit circle.*

A4 *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 $\varepsilon(k)$ as

$$\varepsilon(k) = w(k) - y_{ref}(k). \tag{15}$$

From (14) and (12) we have

$$\varepsilon(k) = w(k) - \phi(k-1)^T\hat{\theta}(k-1) = e(k). \tag{16}$$

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

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

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

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

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

$$|w(k)| \leq a_1 + a_2 \max_{0\leq h\leq k} |\varepsilon(k)|, \quad a_1 \geq 0,\ a_2 > 0.$$

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

$$|u(k-1)| \leq b_1 + b_2 \max_{0\leq h\leq k} |\varepsilon(k)|, \quad b_1 \geq 0,\ b_2 > 0.$$

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

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

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