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 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 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.

I. 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. 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 [1]. 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.

A. 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 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 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 [2], [3], [4], [5], [6].

B. 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 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 ensure stability and system performance, we use a control-theoretic approach for designing the feedback loop. In particular, we use a *direct self-tuning adaptive controller*, the parameters of which adapt on-line to system and workload dynamics, without prior tuning.

Existing systems that apply control theory to computer systems (LotusNotes [7], Apache [2], [8], [9], [3], [5], [6], Squid [10], middleware [11], file server [12]) use non-adaptive controllers that are designed off-line. Other systems [13], [14], [15], [16], [17], [18], [19], [20], [21] 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 workloads. For example, a workload that retrieves data from an internal cache has very different behavior from one that gets data from a disk. In an extended version of this paper [22], we show 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 when the operating range changes. This precludes the use of any of the prior-art mentioned.

There are two recent cases in the literature where an adaptive controller is used to control either application performance for web cache access [23] or CPU utilization in a distributed system [24]. Both those solutions are based on a centralized controller and require modifications of the controlled system. In our case, *no modifications* are required for the target system. In the 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. The only other distributed control loop we are aware of [25] is based on a non-adaptive controller.

II. Specifying performance objectives

As discussed in section I, 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 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.[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 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.

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 three *bands* for throughput sharing, as shown in Table I. According to the specification, the first

[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.

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.

TABLE I

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.

	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%

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.

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 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, $m > i$.

III. 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

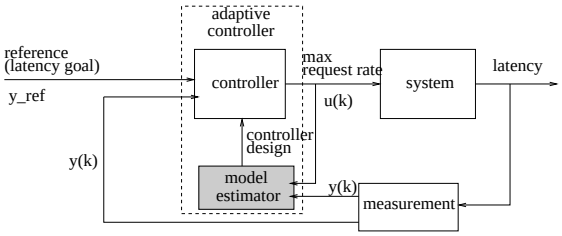

Fig. 1. Feedback loop for client request throttling.

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 [26], cluster file systems [27], or block-based storage [28]. 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 and the latency goals. 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, $y(k)$ is the observed latency of the system averaged over some sampling period, the length of which is specified by a system identification process [22]. The input to the closed-loop system, y_{ref}, is the reference value for $y(k)$. Based on the difference between $y(k)$ and y_{ref}, 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. 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 work-

load 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.

A. Infeasibility of non-adaptive control

A first option for a controller design would be to use classical linear, non-adaptive control, because it is relatively easy to argue about stability and performance of the closed-loop system. Non-adaptive controller design relies on the assumption that the system can be adequately described by one single linear model. In our case, this refers to the function between throughput and latency. Unfortunately, storage systems typically change a lot, either because of the dynamics of multiple concurrent workloads, data being fetched from devices of varying speed (e.g., cache versus disk) or because of changes in the configuration of the system itself. The models for these cases differ significantly [22]. A stable, conservative design would lead to unacceptably long settling times; a more aggressive controller wold lead to instability [22]. We cannot even design a separate non-adaptive controller for each possible operating range of a realistic system, because such operating ranges are not known a priori.

IV. 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. Such controllers adapt on-line to system dynamics in two stages. First, they estimate a system model using an on-line system identification process. Second, they design on-line an appropriate controller for the current system model. In practice, on-line closed-loop design using these two steps may be time consuming and may result in poorly conditioned loops for some parameter values. Instead, we use a *direct self-tuning regulator* [29] 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 1.

A. Analysis of the adaptive closed loop

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 the control law, the adaptive controller first needs to estimate a model for the system that can be turned into a controller. We will show how to do this, by starting from the following generic model.

$$y(k) = s_1 y(k-1) + r_1 u(k-1) + r_2 u(k-2) \quad (1)$$

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 have found that $y(k)$ sampled at 1 second sample intervals captures sufficiently the system dynamics [22]. We have also found, using a number of different model fitting metrics, that a first-order model describes accurately our target system [22]. However, in order to form a direct self-tuning regulator of a first-order system, we need to start from (1) which is a second-order model. The reason for this will be explained in the following paragraph.

The model parameters of (1) are estimated using a *Recursive Least-Squares* (RLS) estimator, an on-line version of the well-known least-squares regression process. To turn this model into a controller, we observe that a controller is a function that returns $u(k)$. If we shift equation (1) one step ahead in time and solve for $u(k)$, we get:

$$u(k) = \frac{1}{r_1} y(k+1) - \frac{s_1}{r_1} y(k) - \frac{r_2}{r_1} u(k-1) \quad (2)$$

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 $k+1$, i.e., it is y_{ref}. Thus, the final control law is:

$$u(k) = \frac{1}{r_1} y_{ref} - \frac{s_1}{r_1} y(k) - \frac{r_2}{r_1} u(k-1) \quad (3)$$

The *stability* of the proposed adaptive controller can be established using a variation of a well-known proof from the literature [29]. 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

```
1   if < MINREQ 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  else
17      u(k) = u*(k)
18  old model = current model
```

Fig. 2. Pseudo-code description of adaptive controller.

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 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, 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 extended version of this paper [22].

B. 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 2, 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* [29]. It checks whether there are enough requests in the last sample interval for it to be statistically significant—MINREQ requests are required (MINREQ $= 6$ in the current prototype). Otherwise, neither the model parameters are modified nor the actuation is computed. To avoid potential system deadlock when all controllers decide not to do any changes (e.g., the system becomes suddenly slow because of a component failure), one random controller in the system does not execute this if-statement. This ensures that at least one control loop is always executed and affects the system.

At every sampling period, the algorithm performs an on-line estimation of the model of the closed-loop system (equation (1)), as described in Section IV-A. That is, it estimates parameters s_1, r_1 and r_2 using least-squares regression [29] 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 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 pays off, as high excitation means a better model and thus 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 $r_1 = 0$. 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.

Finally, the new operating point $u(k)$ is calculated in line 11 using equation (3) 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} [29]. An iteration of the algorithm completes with updating the old model with the new one in line 18.

V. EXPERIMENTAL RESULTS

In this section, we use experimental results to confirm the analytical arguments made in Section IV. We demon-

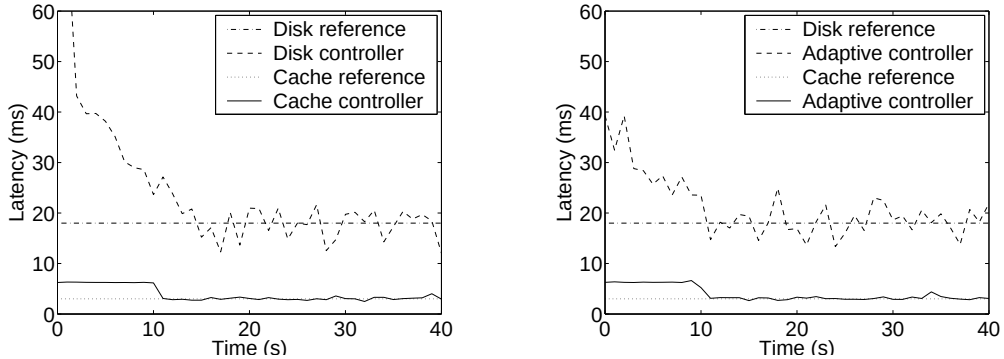

Fig. 3. The performance of the non-adaptive controller on the left and of the adaptive controller on the right. The controllers are enabled at time 10 s. Two workloads are considered, a cache-bound one and a disk-bound one, each with its own latency goal.

strate 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 specified as in Section II.
- 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 [30] 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 a non-adaptive controller that could have been designed off-line for a specific operating range, assuming that the system remains within that operating range (see [22] for details on the non-adaptive controller). Figure 3 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.

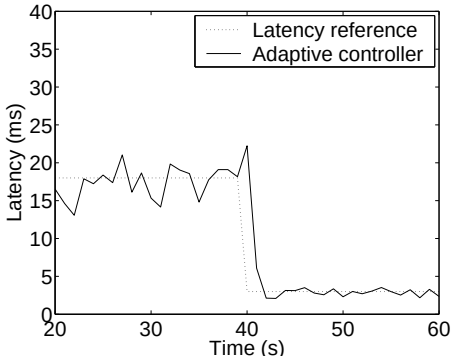

Fig. 4. The performance of the adaptive controller when the workload changes from disk to cache-bound. The latency goal is also changed to demonstrate the adaptability of the model estimation. The workload as well as its latency goal change at time 40 s.

Figure 4 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 s.

Figure 5 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 throughput goals of the two workloads—these reference values represent the aggregate throughput goal for each workload as specified by the throughput bands (Section II), that is achievable with the current system capacity. Before the controller is activated, the throughout goal of workload 2 is exceeded by more

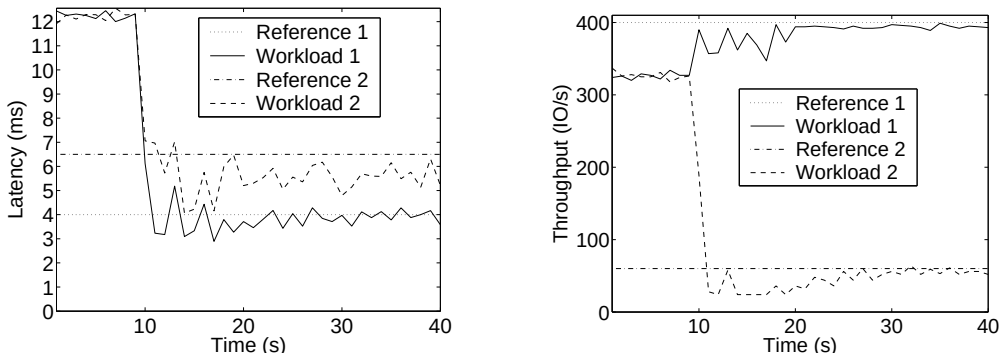

Fig. 5. Latency and throughput isolation when running two cache-bound workloads. The controller is enabled at time 10 s.

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 6 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 s, the system's capacity is doubled by adding an additional server. The data is now striped across both servers[2] 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 2) and the controller changes the throttling performed on workload 2 within 2 s from system reconfiguration.

[2]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.

VI. 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 an adaptive control law is the only appropriate generic way to control a storage system. A 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 workloads cannot be predicted.

In this paper, 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.

Acknowledgments

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

References

[1] 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)*, Monterey, CA, January 2002, pp. 175–188.

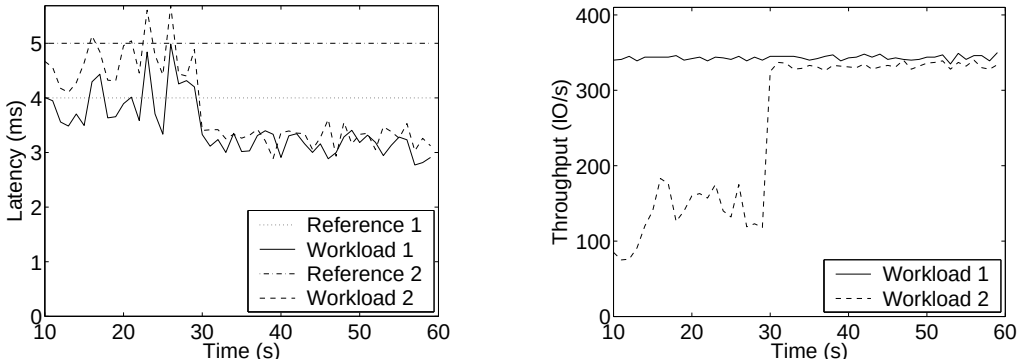

Fig. 6. Latency and throughput differentiation in a dynamic system. Both workloads are cache-bound. At time 30 s, the capacity of the system (number of servers) is doubled.

[2] 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*, vol. 13, no. 1, 2002.

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

[4] 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)*, Providence, RI, May 2003, pp. 184–193.

[5] 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)*, Taipei, Taiwan, June 2001, pp. 51–62.

[6] 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.

[7] 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*, vol. 23, no. 1-2, pp. 127–141, July-September 2002.

[8] T. Abdelzaher, K. G. Shin, and N. Bhatti, "User-level QoS-adaptive resource management in server end-systems," *IEEE Transactions on Computers*, vol. 52, no. 5, 2003.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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)*, Florence, Italy, October 2003, pp. 109–118.

[14] Y. Diao, J. Hellerstein, and S. Parekh, "Using fuzzy control to maximize profits in service level management," *IBM Systems Journal*, vol. 41, no. 3, pp. 403–420, 2002.

[15] Y. Diao, J. Hellerstein, S. Parekh, and J. Bigus, "Managing web server performance with AutoTune agents," *IBM Systems Journal*, vol. 42, no. 1, pp. 136–149, 2003.

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

[17] 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)*, Monterey, CA, June 2003, pp. 461–478.

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

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

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

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

[22] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: Performance Isolation and Differentiation for Storage Systems," HP Laboratories, Tech. Rep. HPL-2004-40, February 2004, http://www.hpl.hp.com/personal/Magnus_Karlsson.

[23] 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)*, Miami Beach, FL, May 2002, pp. 23–32.

[24] 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.

[25] 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)*, London, UK, December 2001, pp. 59–72.

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

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

[28] 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)*, Lihue, HI, May 2003, pp. 169–174.

[29] K. J. Åström and B. Wittenmark, *Adaptive Control*, 2nd ed., ser. Series in Electrical Engineering: Control Engineering. Addison-Wesley Publishing Company, 1995, iSBN 0-201-55866-1.

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