Non-Intrusive Performance Management for Computer Services

Magnus Karlsson[1] and Christos Karamanolis[2]

[1] Enea, Stockholm, Sweden, magnus.karlsson@enea.com
[2] VMware Inc., Palo Alto, U.S.A., christos@vmware.com

Abstract. Networked computer services are increasingly hosted on shared consolidated physical resources (servers, storage, network) in data centers. Thus, some form of resource control is required to ensure contractual performance targets for service customers under dynamic workload and system conditions. This paper proposes a solution for resource control that maximizes the yield of the performance contracts given the available physical resources, while it does not require any modifications to the clients' and the computing services' software or hardware. Our approach achieves this by manipulating the flow of requests into the service by using one or more proxies between the clients and the service.
This paper evaluates Proteus, a prototype implementation of the proposed approach, on two different services: a 3-tier e-commerce system and a networked file service. We show that existing proxies for the two respective protocols (HTTP and NFS RPC) can easily be modified to use Proteus to schedule their requests. Once the modified proxies have been deployed, our approach is transparent to clients and services. Moreover, we show that, in contrast to prior art, our solution (1) is stable when workloads and systems change, (2) automatically tunes itself to different services, (3) can enforce flexible quality of service specifications, and (4) correctly detects and reacts to contention of internal service resources.

1 Introduction

Increasingly, computing services are hosted using clustered architectures, rather than single servers, where a number of distributed physical resources (servers, storage, network) together offer a service. Moreover, service providers and enterprises use shared pools of resources to host multiple customers of a service and/or more than one service. Multiplexing services onto a shared infrastructure allows for on-demand assignment of resources and, thus, improves resource efficiency and cuts management costs.

This paper is concerned with how to manage the performance of a shared computing infrastructure. Negotiated *Service Level Agreements* (SLAs) define contractual performance objectives, such as throughput and response time bounds, and corresponding monetary returns for those performance objectives. The yield derives from the revenue for serving the service workloads, less any penalties for failing to meet contract terms. This paper focuses on the problem of *performance management*, i.e., how to share resources between customers/workloads given the choices already made for admission control and provisioning. These policies usually overbook resources to improve resource utilization and efficiency, assuming a statistical multiplexing of the service demands of different customers. When the total demand exceeds the capacity of the provisioned resources, then a performance management mechanism is needed to share the available capacity, in a way that first isolates workloads from each other and second maximizes the yield obtained from the service given the SLAs in place.

A lot of existing research and commercial systems that provide performance management depend on modifications in the operating system [1, 2], middleware [3], or application code [4]. Clearly, such intrusive approaches are not generally applicable. Thus, a number of non-intrusive approaches have been proposed to intercept and control the workloads as they enter the service infrastructure [5–9]. All these approaches suffer from drawbacks that affect their general applicability.

First, all non-intrusive approaches depend on some form of feedback about the performance delivered to each performance class. The feedback loops of existing solutions are implemented in some ad hoc way, usually employing heuristic algorithms. As a result, there is no guarantee that the system is stable and that it converges to the desirable performance goals when workloads and systems outside the experimental evaluation are used. Second, prior art requires tuning according to the specific service, infrastructure configuration and workload characteristics, something that is ever changing. Third, they may unfairly penalize the performance of workloads given that they do not know who contends for what resources in the infrastructure. For example, if we have ten workloads and one of them has poor performance because it contends for resources with only one other class, then the only way to improve its performance is to reduce the throughput of the contending class. Reducing the others, as usually happens with prior approaches [5–9], will only decrease the total system throughput.

Finally, none of them allows for enforcing *flexible performance goals* that take into account the state of the workloads and resource usage. Prior approaches [5, 7, 8] assume simple static SLAs in the form of a single latency goal that will be guaranteed up to a throughput limit. If demand exceeds the throughput limit or the service cannot provide the latency goal due to workload or system variations, then either no performance guarantees are made [9], the client application is required to throttle back requests [8], or requests are dropped from mainly the performance class with the strictest latency goal [5]. Clearly, such approaches are not acceptable by all applications and services. E.g., dropping requests destined for a disk array is not an option and penalizing the class with the strictest latency goal is unacceptable if that one workload is more important than the others. A more flexible way of specifying performance goals is required.

To address these needs, we propose a new non-intrusive approach for performance management of computing services. Our approach uses standard proxies to intercept service requests before entering the system. These proxies have been modified to use our Proteus library to schedule their requests. Once the modified proxies have been deployed, our approach is transparent to clients and services. Internally, Proteus uses a *control-theoretic adaptive controller* to schedule the requests so that the performance of the service is automatically adjusted according to the specified SLAs. The controller automatically adapts to system and workload characteristics. Thus, it requires no tuning between different services or as the system and workloads change. We prove that the proposed closed-loop design is stable for throughput goals. Last, the controller automatically detects workloads that contend for internal service resources without any prior knowledge about those resources. Thus, control actions do not penalize workloads that do not use bottleneck resources. We report on an prototype of our approach and on experimental results with two diverse computing services: a 3-tier e-commerce system and an NFS service. In both cases, we show that Proteus can achieve all the previously mentioned goals and that the proxies can easily be modified to use Proteus.

2 Overview

This section introduces the basic system model in Section 2.1 and a flexible and generic way of describing performance goals and performance differentiation policies in Section 2.2. In Section 2.3, we show that the performance of a real service varies quickly and often. Thus a static solution or a human in the loop is not an option. Instead, we need an automatic controller that we develop in Section 3.

2.1 System model

We assume that the system consists of a stream of requests dispatched from a set of clients. The requests are processed by a service and returned to the clients in some arbitrary order. Each request can be associated with a *performance class*. A class can be made up of e.g., a specific set of clients or any client using a set of services. A service usually uses a multitude of compute, storage and networking resources. The actual number, layout and performance characteristics of these resources are assumed to be unknown.

In order to be able to isolate and differentiate performance classes, one or more proxies are interposed on the network path between the service and its clients as in Figure 1. The proxies are modified to use our software library to schedule their requests through the API discussed in Section 4. The library consists of two parts: a controller and a scheduler. The scheduler intercepts requests and re-orders or delays them to achieve the partition of throughput capacity corresponding to a configurable *share* setting. For example, with two classes, setting the share to 2:1 means that one class will get 2/3 of the throughput, while the other one gets 1/3. The second scheduler parameter is the *concurrency level* that decides the maximum number of requests inside the system at any given point in time from all the classes. The premise is that the performance (latency and/or throughput) of a class varies with the amount of resources available to execute it. Thus, the scheduler enforces approximate proportional sharing of the service's capacity to serve requests aiming at meeting the performance goals of the different classes. The controller (described in Section 3) will then set these parameters so that the performance goals are met.

The scheduler in our system implements Controllable Start-Time Weighted Fair Queuing (*C-SFQ(D)*) [10], a scheduling algorithm common in computer systems, for four reasons. First, it is computationally efficient; second, being work conserving it results in high resource utilization; third, it is responsive to parameter changes; and fourth, it works in systems with high degree of concurrency.

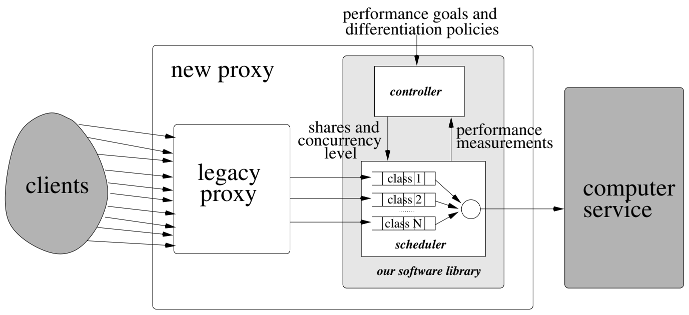


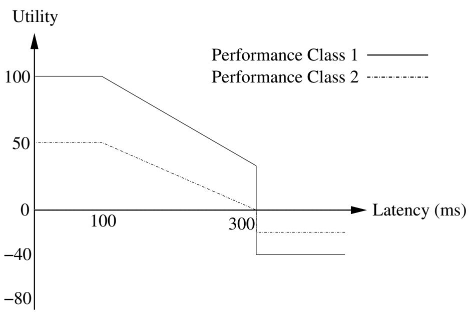
Fig. 1. Our software layer is integrated into a proxy and consists of two parts: a scheduler and a controller. The scheduler controls how a resource is shared by a number of performance classes, while the controller sets the scheduling parameters based on performance feedback.

2.2 Specifying performance goals

In contrast to prior approaches, we provide a way to generically specify any performance goal and performance differentiation policy. A *performance differentiation policy* is used to modify the performance goals when there is not enough capacity in the system to satisfy all the classes' performance goals. The reason for this is that we do not believe that a single policy or goal formulation is good for all possible services, workloads and systems. For a web service with gold, silver and bronze class customers, it might be enough to have a simple priority differentiation policy to prioritize gold over silver over bronze when the system is overloaded. On the other hand, for a shared remote file system in a company, we might want to provide low latency for business critical workloads, but at the same time ensure some slice of the throughput to guarantee forward progress of the backup application. Other situations require completely different performance goals and differentiation policies.

Utility functions [11] provide a way to flexibly specify performance goals and differentiation policies. These are monotonically increasing (for throughput and bandwidth) or decreasing (for latency) functions of one or more performance measurement. The measurements can either be averages or percentiles and there is one utility function per class. The goal of the system is to maximize the total utility obtained, given the utility functions of the classes. In the example of Figure 2, a user of performance class 1 does not want the request latencies to be above 300 ms. In that case the provider should pay 40 monetary units as a penalty. The user is willing to pay for performance above 300 ms, but at the most 100 monetary units for 100 ms. Simpler goals such as "provide at the most 100 ms latency and nothing else" of prior work [5, 8, 9] can be achieved by a utility function that gives x monetary units for 100 ms or less and $-x$ for anything above 100 ms. It is also possible to describe the utility as a function of more than one goal, e.g., utility as a function of both latency and throughput is useful. With these, throughput and latency can be traded off against each other according to user specifications, or we can specify strict goals for both of them.

The performance differentiation policy is captured by the difference between the utility functions of the classes. If one class consistently pays more than another, as is



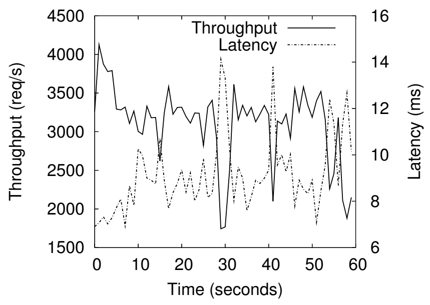


Fig. 2. A utility function example to the left. The right graph shows that a fixed share of the service capacity result in widely varying throughput and latency in our experimental service.

the case in Figure 2 where class 1 always pays more than class 2, we have a priority policy preferring the high-paying class. If the utility functions alternately pay more than each other for each performance level, we have a fair sharing policy, as maximum utility is achieved by alternately giving each class a little bit more of performance. A plethora of other differentiation policies are possible. This way, the users of our system are not locked into a specific differentiation policy or performance goal specification as in prior art [5, 8, 9]

Throughout the paper we will mostly talk about high level performance goals such as “provide 100 ms request latency” or “give me between 100 and 400 req/s” combined with high level performance differentiation policies such as “priority”, “fair share” and “best effort”. The reason for this is that these are easier both to understand and to validate than arbitrary chosen utility functions. Note, that these performance goals and differentiation policies are all specified with utility functions and, as we will see in Section 3, utility functions are the only performance specification that is used internally by our library to achieve our goal of flexibility.

2.3 Need to vary shares and concurrency

A key problem is that, in the general case, certain share and concurrency level assignments do not result in steady and predictable performance, because of the dynamic nature of workloads and systems. The right graph in Figure 2 depicts the example of a typical workload of a remote file service. Even in this static example where the system does not change (there is a fixed number of clients running the same workload on a fixed set of resources), a fixed share and concurrency level (80% and 16 in this example) for one class results in widely oscillating throughput and latency. If for example, we would like to provide exactly 2,000 req/s to a class, that would mean 100% of the share around 30 s. and around 50% of the share at the beginning of the execution. Clearly, the shares need to change dynamically. A solution that does not dynamically vary the shares [5, 8, 9] can fundamentally not provide a flexible performance differentiation policy, as the sharing of the system is fixed.

The concurrency level also needs to vary dynamically. When the concurrency level is high, the system is used efficiently as it provides the highest total throughput. But the

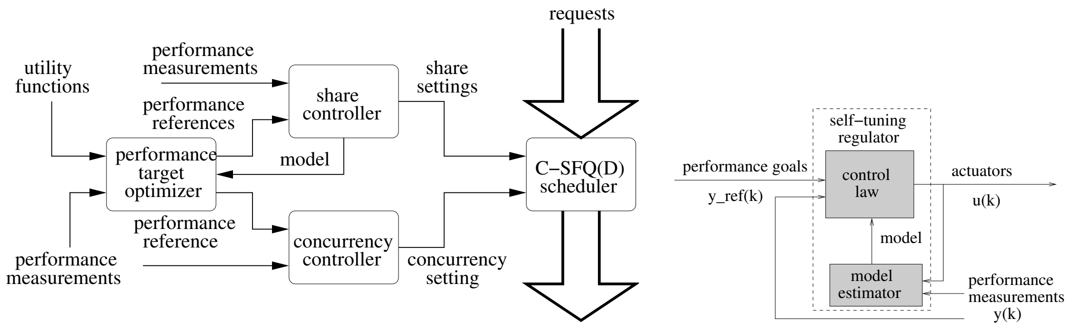


Fig. 3. Overview of our solution to the left and a self-tuning regulator to the right.

more requests inside the service, the higher the request latencies will be, given a fixed amount of resources. There is thus a trade-off between these goals. The concurrency level that achieves the best trade-off varies with for example, what other requests are inside the system, what resources are used and resource lay-out. As this changes, the concurrency level also needs to vary dynamically.

3 Adaptive Resource Control

In this section, we will design a controller for automatically and dynamically setting the shares and the level of concurrency of the scheduler. We will use *adaptive control theory* [12] described in Section 3.1, as it provides a well understood methodology for designing closed-loop systems that are stable, efficient and meet their goals. On a high level, an adaptive controller has two sets of inputs, a set of performance measurements from the system and a set of desired performance references. The goal of the controller is to get the performance measurements to equal the desired references by adjusting one or more system parameters. In order to achieve this, one of the most important tasks for it is to internally estimate a model of how the system parameters affect the measurements.

Figure 3 shows an overview of the control architecture we propose to provide adaptive resource control. It consists of one controller for setting the shares and one for setting the level of concurrency. The share controller in Section 3.2 sets the shares of the performance classes so that their performance goals are met. The concurrency controller in Section 3.3 adjusts the level of concurrency as to achieve the best trade-off between high capacity and meeting the strictest latency goal. An optimizer, described in Section 3.4, computes the performance targets for the classes, using the utility functions of the classes and the performance model produced in the share controller. These performance targets are then fed as references to both controllers, that they can adjust the shares and concurrency so that the classes achieve their performance targets. The targets are continuously modified so that they are always possible to achieve. The extension to distributed proxies is described in Section 3.5.

3.1 Adaptive optimal control theory

In order to explain the controllers, we need to introduce some notation. Assume that there are N measurements of interest made in a system, and let $y(k)$ be a column vector of the N measurements sampled at time k. These measurements are statistical metrics (e.g., average or percentile) computed over the sample period $(k-1, k]$. The elapsed wall clock time between $k-1$ and k is called the *sample interval* and is assumed to be constant over time. Let column vector $y_{ref}(k)$ denote the desired values for the N measurements at time k, and let $u(k)$ capture M actuator settings at time k. An *actuator* is a system parameter that can be dynamically set. For example, it can be the resource share of each performance class. Informally, the problem is to have the N measurements converge to the specified desired values by dynamically setting the M actuators in the system. This can be formalized as the following optimization problem:

$$minimize\ J(u(k)) = ||W(y(k+1) - y_{ref}(k+1))||^2 + ||Q(u(k) - u(k-1))||^2 \quad (1)$$

$W \in \mathbb{R}^{N\times N}$ and $Q \in \mathbb{R}^{N\times M}$ are positive-semidefinite weighting matrices. W captures the importance of meeting the reference values for different measurements, and Q reflects the penalties for large actuation changes. This objective function J is expressed as a function of $u(k)$, as the latter is the only input that can affect the state of the system. Note that $J(u(k)) \geq 0$ in all cases. It becomes zero when all the measurement references are met and there is no change to the actuators between consecutive intervals, i.e. the desired state.

The advantage of the problem formulation in (1) is that there are well-understood controllers that can be applied to solve it. More specifically, *optimal control* [13] is a field of control theory, in which control goals are formulated as optimization problems. Controllers designed following this approach are constructed so that they aim for the optimal solution to (1), while guaranteeing that the system is stable and converges fast.

There are a number of requirements that such a controller should meet. It has to be computationally efficient—as it needs to perform on-line control of the system. The controller should require little or no knowledge of the target system and should need little or no manual tuning before being applied. It should also quickly adapt to changes to the system and its workloads and the closed-loop should always be stable.

We propose using a certain type of adaptive controllers called *Self-Tuning Regulators* (*STR*) [12], that can enforce on-line optimization of optimal control problems while they satisfy the above practical requirements. The structure of an *STR* is shown in the right diagram of Figure 3. It consists of an *estimator* module that on-line estimates a model that describes how actuator setting affect the measurements. That model is then used by a *control law* that decides how to set the actuators so that (1) is minimized while guaranteeing stability.

3.2 Shares controller

Meeting performance goals and providing flexible performance differentiation by manipulating the share of resources each class gets, can be formulated as an optimization problem using (1). In this case, the vector $y(k)$ refers to the performance measurements of each of the N performance classes, sampled at time k. These measurements can refer to either request latencies, throughput, or both. Vector $y_{ref}(k)$ represents the desired performance. When these targets cannot be met, $y_{ref}(k)$ will be adjusted by the optimizer in Section 3.4 according to the performance differentiation policy. $u(k)$ captures the individual share settings of each of the N performance classes. Minimizing $J(u(k))$ in this case means that the system converges to the performance targets for the N classes, while minimizing the necessary changes in share settings.

For model estimation in the *STR* we use a linear model of the following form:

$$y(k) = \sum_{i=1}^{n} A_i y(k-i) + \sum_{i=0}^{n-1} B_i u(k-i-d_0) \tag{2}$$

Here A_i and B_i are the model parameters. Note that $A_i \in \mathbb{R}^{N \times N}$, $B_j \in \mathbb{R}^{N \times M}$, $0 < i \leq n$, $0 \leq j < n$, where n is the *model order* that captures how much history the model takes into account. Parameter d_0 is the delay between an actuation and the time the first effects of that actuation are observed. The diagonals of A_i and B_i state what performance a class receives from a given share setting, while the anti-diagonals describe the effect changing the shares of the other classes have on a given class. Thus, the first advantage of the model is that it models how the classes are contending for resources. To correctly capture and react to this was one of our goals of the architecture. A second advantage is that it is simple and generic, a prerequisite for wide system applicability. A third advantage of this model is that it captures the dynamics of the system. This is important as higher share many times results in worse performance for a short period before the performance gets better. The underlying reason for this is e.g., warming up of caches. If the model in the controller did not take this into account, it might increase the share even further as a response to the worse performance observed shortly after the change, leading to oscillations and in the worst case to instability. The final advantage with this model is that it does not assume that all requests in all classes consume the same amount of resources. One class can have a higher entry in B_i than the others reflecting requests that take longer to serve. Prior work [5, 9] does not take any of these four effects into account.

We know that the relation between actuation and performance is not always linear. For example, latency is inversely related to the share. However, even in the case of non-linear metrics, a linear model is a good enough local approximation for the controller, as it usually only makes small changes to actuator settings. But the estimation can be improved by inputting the inverse of the latency, which we do for the rest of the paper. The advantage of using linear models is that they can be estimated in computationally efficient ways, that result in tractable control laws and there are stability proofs for them.

The unknown model parameters A_i and B_i are estimated using *Recursive Least-Squares* (RLS) estimation [12]. This is a standard, computationally fast estimation technique that fits (2) to a number of measurements, so that the sum of squared errors between the measurements and the model is minimized. RLS is able to estimate even the performance correlation between the classes (the anti-diagonals of A_i and B_i).

The control law is a function that, based on the estimated system model (2), decides what the actuator settings should be so that objective function (1) is minimized. One of the reasons we specifically chose (1) and (2) is that $u(k)$ can be computed using a computationally efficient closed-form expression. The full derivation can be found in Karlsson *et al.* [14].

From a systems perspective, the important point is that the control law provides a computationally efficient way to calculate $u(k)$ which can be performed on-line. This *STR* requires little system-specific tuning as it uses a dynamically estimated model of the system, the control law automatically adapts to system and workload dynamics.

It has been shown [12] that this *STR* is stable iff the following is true of the system: (i) the initial delay (d_0) is known and bounded; (ii) the system is minimum phase[3]; (iii) the signs of the triangular elements of B_0 are known; (iv) the upper bound on the order (n) of the system is known; and (v) the measurements are linearly related to the actuators. If true, our control-loop is stable and the performance converges to the performance goals in steady state. Independently of the system we run on, we know that the diagonal of B_0 is positive for throughput and negative for latency, and that throughput is linearly related to the share, while latency is nonlinearly related. The other parameters are system dependent and hold for our experimental systems. This means that the controller is provably stable for throughput goals and for latency goals when the changes to the system and workloads are small (as linearity is a good approximation). It is a topic of future work to analyze the stability of the loop for latency goals under arbitrary workload and system changes.

3.3 Concurrency controller

For the concurrency level controller we are going to use the same control law and estimator as for the share controller, though with different measurements and actuators in $y(k)$ and $u(k)$, respectively. $u(k)$ will now contain the concurrency level as that is what we desire to set. The trade-off in setting this is between high total throughput of the system and being able to meet the tightest latency goal. Generally, the higher the concurrency, the higher the total throughput (until we get into overload). The end-to-end latency consists of the sum of two parts: the waiting time inside the scheduler ($w(k)$) and the service time[4] ($\sigma(k)$) inside the service itself. As the service times also monotonically increase with an increase in concurrency level, we can achieve the best total throughput while still being able to satisfy the tightest latency goal by having the system operate around a service time that will provide that end-to-end latency goal. Thus, $y_{ref}(k)$ of the concurrency controller should contain the service time it should aim for, that is the tightest latency goal minus the wait time $w(k)$ for that class. The corresponding measured service time is put in $y(k)$. This computation of $y_{ref}(k)$ is performed by the performance target optimizer and fed to this controller. The control law is then used to compute the concurrency level in $u(k)$. Note, that we do not need to care about the other classes' latency goals here, as they will be met by adjusting the shares with the share controller.

The stability conditions are the same as for the share controller. Note that these two stability proofs are only valid when the two controllers are applied to the scheduler

[3] This basically means that new actuator settings have precedence over old actuator settings.

[4] Queuing time plus processing time inside the service.

```
1  $s_{ip} = 0 \;\; \forall i \in classes, \;\; \forall p \in proxies$
2  $U_i^{curr} = -\infty \;\; \forall i \in classes$
3  while $\sum_{i=1}^{N} s_{ip} < 1 \;\; \forall p \in proxies$
4      for all performance classes $i$ and proxies $p$
5          find smallest integer $d_{ip} > 0$ for which $U_i(\hat{X}_p(s_{ip} + d_{ip}/T_{tot,p}(k))) > U_i^{curr}$
6          if no such $d_{ip}$ or $d_{ip} + \sum_{i=1}^{N} s_{ip} > 1$
7              $d_{ip} = 1$
8      end for
9      find $j$ and $p$ that maximize $U_j(\hat{X}_p(s_{jp} + d_{jp}/T_{tot,p}(k)))/d_{jp} - U_j^{curr}$
10     $s_{jp} = s_{jp} + d_{jp}/T_{tot,p}(k)$
11     $U_j^{curr} = U_j(\hat{X}_p(s_{jp}))$
12 end while
13 $y_{ref,p}(k) = \hat{X}_p(s_p) \;\; \forall p \in proxies$
```

Fig. 4. Pseudo-code of the performance target optimizer used to compute $y_{ref,p}(k)$.

individually. It is a topic of future work to analyze the stability of the two loops together. Currently, we make sure that the concurrency controller's effect on the share controller's measurements are small by increasing Q in (1) for the concurrency controller, making it less aggressive than the share controller.

3.4 Performance target optimizer

As noted in the beginning of Section 3, the utility functions we use to describe performance goals and differentiation policies need to be translated into a setting of the references ($y_{ref}(k)$) for the controllers at each time interval. The problem is to find the share settings or resource partitioning that maximize the utility of the system given the current service capacity. Fortunately, we can derive a function relating shares to utility, as utility (U) is a given function of performance and RLS estimates a model correlating performance with shares inside the share controller. For notational convenience, we will here denote that model from (2) as $\hat{X}_p(s)$ where s is a vector of shares for the N classes and p indicates what proxy this is from. We have so far only presented the solution for one proxy, so this p can be ignored for now. But in the next section, we will present a distributed solution that will use multiple proxies.

Throughput is linearly dependent on the shares, so this optimization problem can be easily solved using linear algebra. But for latency, the model $\hat{X}_p(s)$ has a nonlinear dependency on the shares (in contrast to the latency inverse that was entered). Thus we cannot solve this optimization problem using standard linear algebra. Instead, we use a greedy optimization heuristic depicted in Figure 4 that works as follows. Each performance class is initially given 0 in share (s) in line 1 and the current utility (U_i^{curr}) provided by class i is initialized to minus infinity in line 2. We then find the smallest increase in share that will provide an increase in utility for each class in line 5. (The granularity of share increases is set to the share that one request consumed on average during the last interval, i.e. $1/T_{tot,p}(k)$.) The reason we do this is to deal with utility functions that have sections that are flat (as in Figure 2 for latencies above 300 ms). If there is no such share increase or this increase would be larger than giving that class the rest of the system capacity, the share increase is set to its minimal value in line 7. In line 9, we then find the class that provides the largest increase in utility per share by giving it a share corresponding to the share increase computed in line 5. This class is then given this amount of extra share in line 10, and the process is repeated until the sum of all shares given equals 1. Finally in line 13, we enter the computed shares into the estimated model to compute the latency values to put into $y_{ref}(k)$ for the share controller. The concurrency controller is fed the strictest latency goal found in $y_{ref}(k)$ minus the queue wait time for that class. Note, that changing the reference values does not affect the stability of the controllers, only the time it takes for them to reach the desired goals.

3.5 Distributed proxies

A solution with multiple distributed proxies can be employed in front of the service when there are multiple entry points or when the request rate is so high that one proxy cannot handle it. In this case, the controllers and the scheduler are run on each proxy using local information. However, the optimizer is only executed on one node using measurements gathered from all the distributed proxies, thus it becomes a global optimizer for all the proxies. The same algorithm as in Figure 4 is used, but now there are multiple proxies so the algorithm will iterate over all proxies. That way, we now loop through line 3 to 12 until we have used up all shares on all proxies. Once the algorithm terminates, the individual references for each proxy are distributed to the local controllers on each proxy. The amount of floating point numbers transmitted over the network per sample period is $(4N + 2N^2)(P - 1)$, where N is the number of performance classes and P is the number of proxies. The quadratic term stems from transferring the model and the linear term from the measurements and results. The physical location of this global optimizer is statically defined at start up.

4 Implementation and API of Proteus

Our implementation of the controllers, optimizer and scheduler is called `Proteus`. We have implemented it as a library weighting in at slightly over 10,000 lines of C++. Proteus exports the API tabulated in Table 1. It consists of two parts. The first part is used to initialize the library and register and unregister performance classes and their goals. The second part of the interface is related to the handling of application requests. Whenever a request arrives at the proxy, the request is queued in the proxy and is registered with `prRequest()`, which returns a unique handle for the request. Similarly, when a reply to a request goes through the proxy, the reply is also registered with `prReply()` using the handle of the corresponding request. Proteus specifies the next request to be submitted to the backend service in either of two ways, depending on whether the proxy is implemented as an event-driven or a multi-threaded program. Event-driven proxies register a call-back function using `prInit()`. Whenever one of the Proteus functions are called, Proteus may invoke the call-back with the handle of the next request to be sent out. Threaded proxies, instead, may use a thread to call `prDequeueRequest()`

Function name	**Description**
void **prInit**(cb1, cb2)	Initializes Proteus and registers application callbacks for sending the next request (for event-driven proxies) and dropping requests.
ClassID **prClassInit**(PerfGoal)	Registers a class and its performance goals. Returns a unique ID for this class.
void **prPerfGoal**(ClassID, PerfGoal)	Changes the performance goal of the class identified by ClassID.
void **prExecCtrl**()	To be called repeatedly by the proxy. Once every sample interval, a call to this function will execute the controllers.
QoSReqID **prRequest**(AppReqID, ClassID)	Register an incoming client request to belong to class ClassID. AppReqID is the proxy's request ID. Returns a unique identifier for this request.
void **prReply**(QoSReqID, AppReqID)	Registers a reply with Proteus. The reply is uniquely identified by the <QoSReqID,AppReqID> tuple.
<QoSReqID,AppReqID> **prDequeueRequest**()	Synchronous call to dequeue the next request for submission to the system. QoSReqID is -1 if there is no request to send. Used by threaded proxies.
void **prRemoveRequest** (QoSReqID, AppReqID)	Explicitly remove a pending request from Proteus.

Table 1. The API of Proteus.

to obtain the handle of the next request to be sent to the service or return an error if there is no request for delivery.

The running time of Proteus is around 150 μs on our machines, a low overhead to incur at every sample interval. The sample interval is 1 s which we found to work well empirically. The overhead of Proteus incurred when registering requests and replies is negligible.

5 Evaluation

In this section, we present our prototype implementation Proteus and experimental results showing that it can achieve the goals stated in the introduction.

5.1 Experimental methodology

Our evaluation demonstrates the effectiveness of Proteus using two different network services: a 3-tier e-commerce system and an NFS service. The 3-tier system consists of a web server, two application servers and one database server. They are hosted on separate server blades, each with two 1 GHz Pentium III processors, 2 GB of RAM, one 46 GB 15 krpm SCSI Ultra160 disk, and two 100 Mbps Ethernet cards. The web server is Apache version 2.0.48 with a BEA WebLogic plug-in. The application server is BEA WebLogic 7.0 SP4 over Java SDK version 1.3.1 from Sun. The database client and server are Oracle 9iR2. All three tiers run on Windows 2000 Server SP4. The site hosted on the 3-tier system is the Java PetStore[5].

[5] http://java.sun.com/developer/releases/petstore/

The workload applied to this system mimics real-world client behavior on shopping sites. These clients log in, browse and search for products, put products in their carts, and sometimes checkout the cart which gives rise to credit card verifications, adjustment of inventory, etc. The response latencies vary between 10 ms and 700 ms for the various operations. The workload also captures the corresponding time scales and probabilities these occur with and is generated using `httperf` on a separate machine. Proteus has been integrated into `tinyproxy` v1.6.3 that intercepts the traffic between the client machine and the 3-tier system. Only 15 lines of code had to be added to tinyproxy to use Proteus. For the experiments in the rest of this section, we generate 80 concurrent client sessions and we consider 2 performance classes of 40 clients each, unless noted.

We generate three workload patterns in order to stress the system. `Smooth` keeps the number of clients and their shopping behavior steady with a minimum of changes. In `Ramp`, more clients are gradually added as in TPC-W. After 80 is reached, they gradually start to check out more things which gives rise to even higher load on the system, especially on the database tier. The third pattern, `Step`, makes the same changes as ramp but all at once. This change is repeated in a square wave pattern with a change occurring every 30 s. Note that it is not possible to perform this change instantaneously, as clients must add products to their carts before they can check them out. Thus, the step takes 3-4 s in practice.

Second, we use an NFS file service consisting of five blades with the same hardware specification as the ones above. All blades run Linux 2.4.20. Two blades are used as clients, two as servers and one as an NFS proxy. Both clients and servers use NFS v.3 with asynchronous writes. In order to stress the system, the clients make random reads and writes to individual files on the NFS servers. The full data set is large enough that it does not fit in the in-memory cache of the file server. The traffic between clients and servers is intercepted by a user-space NFS v.3 proxy. Only 20 lines of code were added to integrate Proteus into this proxy. There are 16 concurrent client threads generating requests on each client node. Each node belong forms a different performance class.

5.2 Comparison against prior art

First we will compare Proteus to prior art. We will only compare against non-intrusive techniques as another non-intrusive technique [5] already showed it was comparable to the best intrusive ones. Of the non-intrusive techniques, we did not consider techniques that cannot provide performance isolation [7, 9], nor do we think it is fair to compare a technique [6, 8] not designed for 3-tier systems or NFS servers. This leaves us with Quorum [5] designed for Internet services. We implemented Quorum according to the algorithmic pseudo code reported in their paper. Their scheduler and dropping module was used instead of *C-SFQ(D)* and their controller replaced ours.

In order to make the comparison fair, we will only consider the case which Quorum was designed for; a 3-tier system when there is a combined latency and throughput goal per class and achievable goals. Quorum adjusts the concurrency dynamically based on the latency goal, while the shares (called weights in Quorum) are static and set off-line according to the ratio between the throughput goals. Class one (C1) has a latency goal of 200 ms and a throughput goal of 60 req/s. Class two (C2) has a latency goal of 5,000 ms (effectively best effort so that we can ignore it for the comparison) and a

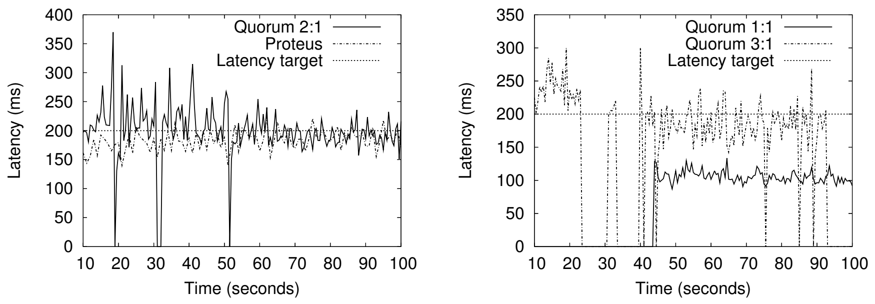


Fig. 5. A comparison against Quorum on the 3-tier system. C1 has a 200 ms latency goal and C2 has a best-effort goal and is not shown in the graphs. Quorum shows unstable behavior when the workload is changing between 10 s and 60 s. As can be seen in the right graph, the stability of Quorum is also dependent on the throughput goals used to statically set the shares.

throughput goal of 30 req/s. The static share ratio is then set to $60 : 30 = 2/1$ in the Quorum scheduler. The workload used is `ramp` as we want to evaluate how Quorum deals with change.

The left graph in Figure 5 shows the provided latency over time for Quorum and for Proteus. A latency of zero means that there were no requests at all during that interval. From this figure we can see that Quorum oscillates widely and misses the latency target during the period the `ramp` workload changes between 10 s and 50 s. Quorum exhibits instabilities around 20, 30 and 50 s when the dropping module of Quorum drops all requests from C1. Proteus with the same latency and throughput goals, on the other hand, has no problem with the changing workload and varies smoothly slightly under the latency target most of the time. After 60 s, when the workload and system is not changing, Quorum and Proteus are comparable.

If we then change the throughput goals to 60:60 (a 1:1 share ratio) or 60:20 (a 3:1 share ratio) we expect Quorum to provide similar latency measurements as for the 60:30 case, as the throughput goals are still easily going to be met (the total capacity is around 150 req/s) and therefore have no impact on the system. But this is not the case as seen in the right graph of Figure 5. Quorum oscillates widely when the share ratio is 3:1 and shows complete starvation for 1:1 during the ramp up, then undershoots the goal by a wide margin once the workload stabilizes. One of the reasons for this is that shares have an impact on not only the throughput but also the request latencies as seen in Figure 2. Proteus, on the other hand, automatically adjusts the shares to produce the desired performance goals. Proteus behaves the same in all these three cases as the references produced by the optimizer are the same in all three cases, because the throughput goals are always met. Clearly, the shares need to be set dynamically in response to both latency and throughput goals and measurements, and not set statically according to the throughput goals only. Moreover, if we would like to support flexible performance differentiation policies and react to internal resource contentions as in Proteus, being able to vary the shares is imperative.

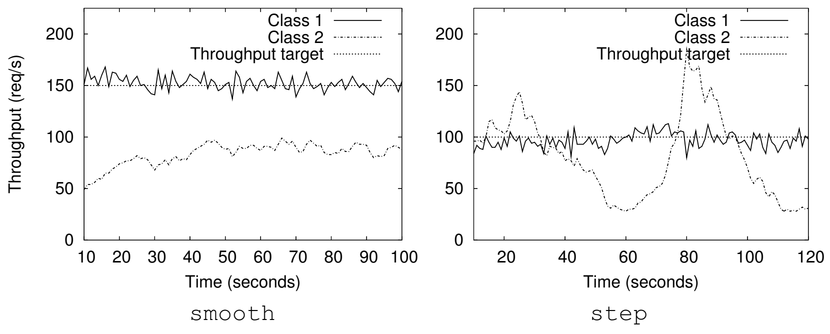


Fig. 6. Proteus on the 3-tier system with throughput goals. C2 has a best-effort goal.

5.3 3-tier e-commerce system

For the rest of the evaluation we will evaluate how well Proteus can achieve the objectives stated previously. That is, to be able to provide flexible performance goals and differentiation policies, show that it works between systems without any system-specific modifications or tuning, it is stable, and show that Proteus can successfully detect and deal with workload and system changes as well as contention for unknown internal service resources.

In this section, we will focus on the 3-tier system. We will start to show that Proteus works for throughput goals with a strict prioritization between the classes. C1 has higher priority than C2. The throughput target of C1 and C2 is given in the graphs and C2 gets any spare capacity. In this case, the goal of C1 can be met for the workloads shown in Figure 6. The throughput does vary a little around the performance target, because the total throughput of the 3-tier system is not constant as was shown in Figure 2.

Figure 7 shows that we can also enforce latency goals. In these experiments, C1 and C2 have the same latency goal, but C1 has higher priority. If it is not possible to meet both class goals, then the latency goal of C1 is met at the expense of C2. To introduce even more variation from the previous experiment, if both goals can be met, then any remaining resources are shared fairly resulting in better latency for both classes. We can see from the graphs that this indeed is the case for all three workloads: for `smooth`, C1 gets a latency around its requested target value, while C2 gets only what is possible; with `step` we can also see that when the capacity is high enough to satisfy both classes, they equally share the excess capacity (with the exception of the initial start up period).

The previous experiments have only considered absolute performance goals with prioritization. However, any other performance goal or performance differentiation policy is possible as long as it can be described by utility functions. Figure 8 shows results for proportional performance goals: in the left graph, throughput is shared 2/3 to C1 and 1/3 to C2; and in the right graph, C1 has to have half of C2's latency. The `ramp` workload is used for both experiments. We see that Proteus effectively also enforces proportional performance goals. Note that C2's latency varies more than C1's. The reason is that the higher the latency, the more sensitive it is to even slight throughput changes. When few requests are let into the system from a class, admitting one more

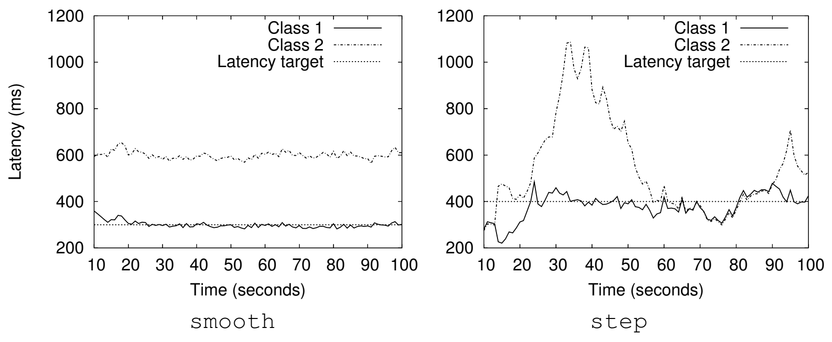


Fig. 7. Proteus on the 3-tier system with latency goals. C1 has higher priority than C2.

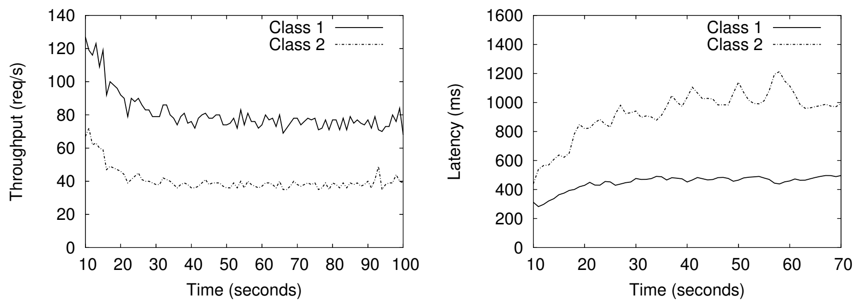


Fig. 8. Demonstrating the enforcement of proportional performance goals in the 3-tier system with `ramp`. On the left, C1 gets 2/3 of the throughput; and on the right, C1 has half the latency of C2.

or less will have a large impact on the average latency measured for that class. One the other hand, when there are many requests from one class, one more or less makes little difference in the measured latency.

To show that Proteus can handle more than two performance classes, we ran an experiment with 4 classes. The results are shown in Figure 9. Each class has a throughput goal of 70 req/s. Priorities are set as C1 $>$ C2 $>$ C3 $>$ C4. We use the `ramp` workload to show how Proteus reacts to changes. As specified, a class receives some throughput only if there is spare capacity after the goals of higher priority classes have been met. C4 is the first one to get throttled back to 0 req/s, followed by C3. Then C2 is scaled back to around 45 req/s, while C1 receives its requested 70 req/s throughout the run.

5.4 A shared NFS service

To demonstrate that Proteus can be effectively used in different systems, we applied the same library used for the 3-tier system to enforce performance differentiation in a shared NFS service. The two services have workloads with different types of requests, response latencies that are orders of magnitude different, and internal structures that are

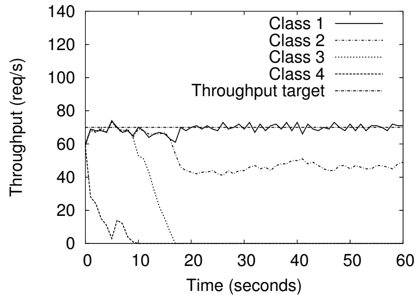
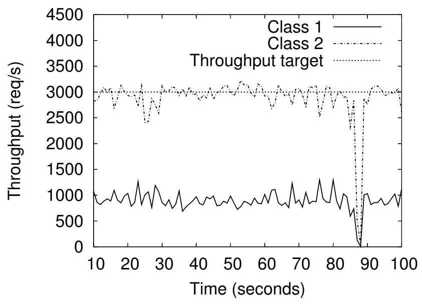


Fig. 9. Showing that Proteus can handle more than 2 performance classes. Here is an example with 4 classes with strict priority amongst them.



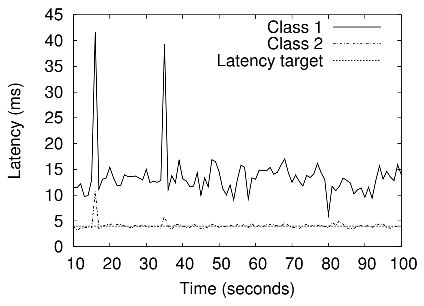


Fig. 10. Using Proteus to meet performance goals in an NFS server. Throughput goals are shown on the left and latency goals to the right.

representative of different types of resource contentions. For example, the application server in the 3-tier system is mostly CPU-bound, while the NFS servers are disk-bound.

Figure 10 demonstrates performance differentiation in the NFS service. Class 1 is the high priority one, while class 2 is best effort. Note, that the performance of the NFS system varies more than the performance of the 3-tier system. The dip in throughput right before 90 s is due to the `kswapd` daemon in Linux being invoked to write pages to disk. During those 1-2 s the throughput of the NFS server is close to zero.

To show that Proteus can successfully detect and deal with changing internal bottleneck resources due to workload variations and/or resource failures, we conducted an experiment where the two classes are accessing different NFS servers during the first 30 s. After that, class 2 switches to accessing the same server as class 1. In this experiment we mounted the NFS partitions in synchronous mode to be able to load the servers more, hence the lower performance numbers. The results of this experiment are shown in Figure 11 for throughput goals. Before time 30 s, both classes are getting their desired 150 req/s as they are accessing disjoint servers. But at time 30 s, they start to share the same server, which cannot meet the goals of both workloads. This is detected by the model estimator in the share controller, and the model starts to show that the performance of the two classes is now correlated. This is taken into account by the op-

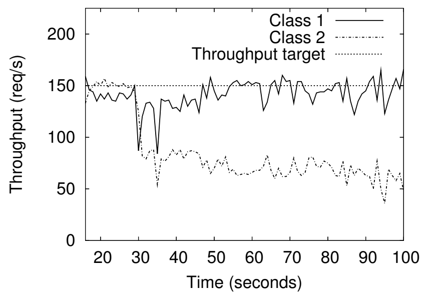


Fig. 11. Proteus can adapt to dynamic resource contention inside the system. At time 30 s, Class 2 switches from its own NFS server to accessing the same server as Class 1.

timizer that adjusts the performance target of the low priority C2 downwards so that the target of the high priority C1 can be met. This takes Proteus only a few seconds.

6 Related Work

During the last few years, there has been a surge of research on automatic performance management of systems. For on-line management problems, most existing solutions consider some form of feedback approach. Many of them use classical non-adaptive controllers [4, 15–18], which for most practical purposes are not adequate in computer systems due to their ever changing characteristics [7, 8, 19]. Some approaches have addressed this problem by proposing some form of adaptive controller designed in an ad-hoc manner [6, 7, 9, 20–22]. Given that the analysis of the controller is based on empirical data, it is unknown whether the resulting controller is stable and performs well beyond the generally narrow experimental evaluation performed. Moreover, they are designed for a specific service and many require modifications to the target service.

To the best of our knowledge, there are six published papers that use adaptive controllers designed using a formal control-theoretic approach to achieve performance goals in computer systems. All of them are intrusive, target a specific system, cannot detect contention between performance classes, and have a static performance differentiation policy, if any at all. Proteus addresses all of these problems. Lu *et al.* [23] constructed an *STR* to satisfy absolute latency goals in web servers by partitioning the cache space. Wu *et al.* [24] used a dual *STR* to control the hit ratio of a web cache by allocating cache space to different users. Karlsson *et al.* [8] used an *STR* to achieve latency differentiation in a clustered file-system. Zhu *et al.* [25] use a adaptive pole placement controller to set the CPU resources given to a performance class in a web server, while Wei *et al.* [19] use an adaptive fuzzy controller to guarantee latency goals in web servers. Finally, Lu *et al.* [26] used another type of adaptive controller called model-based predictive controller to satisfy end-to-end latency bounds in distributed real-time systems. The controller we use here is of the same general type (*STR*) as the first three approaches, though they only design single input and single output controllers and do not use optimal control.

7 Conclusions

In this paper, we proposed a solution for performance management that maximizes the yield of the performance contracts given the available physical resources, while it does not require any modifications to the software or hardware of the computing services or the clients. Our approach achieves this by manipulating the flow of requests into the service by using one or more proxies between the clients and the service. In contrast to prior art, our solution is stable, runs on different services and can enforce flexible performance goals.

We implemented a prototype of our design called Proteus, that was evaluated on two systems, a 3-tier e-commerce system and an NFS file service. We show that existing proxies for the two respective protocols (HTTP and NFS RPC) can easily be modified to use Proteus to schedule their requests. Once the modified proxies have been deployed, our approach is transparent to clients and services. Proteus ensures that both services effectively conform to the SLAs of multiple competing workloads and enforces flexible performance specifications. It adapts within a few seconds to system and workload dynamics, is more stable than prior art, and automatically detects contention on internal service resources to improve overall resource utilization. We also show that it can be used on both systems without any tuning between them.

References

1. Shenoy, P., Vin, H.: Cello: A Disk Scheduling Framework for Next Generation Operating Systems. In: International Conference on Measurement and Modelling of Computer Systems (SIGMETRICS), Madison, WI (1998) 44–55
2. Voigt, T., Tewari, R., Freimuth, D., Mehra, A.: Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In: USENIX Annual Technical Conference, Boston, MA (2001) 189–202
3. Shen, K., Tang, H., Yang, T., Chu, L.: Integrated resource management for cluster-based internet services. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI), Boston, MA (2002) 225–238
4. Abdelzaher, T., Shin, K.G., Bhatti, N.: Performance guarantees for web server end-systems: A control-theoretical approach. IEEE Transactions on Parallel and Distributed Systems **13**(1) (2002) 80–96
5. Blanquer, J., Batchelli, A., Schauser, K., Wolski, R.: Quorum: Flexible Quality of Service for Internet Services. In: USENIX Symposium on Networked System Design and Implementation (NSDI), Boston, MA (2005) 159–174
6. Chambliss, D., Alvarez, G., Pandey, P., Jadav, D., Xu, J., Menon, R., Lee, T.: Performance virtulization for large-scale storage systems. In: Symposium on Reliable Distributed Systems (SRDS), Florence, Italy (2003) 109–118
7. Kamra, A., Misra, V., Nahum, E.: Yaksha: A Self-Tuning Controller for Managing the Performance of 3-Tiered Web sites. In: International Workshop on Quality of Service (IWQoS), Montreal, Canada (2004) 47–56
8. Karlsson, M., Karamanolis, C., Zhu, X.: Triage: Performance isolation and differentiation for storage systems. In: International Workshop on Quality of Service (IWQoS), Montreal, Canada (2004) 67–74

9. Lumb, C., Merchant, A., Alvarez, G.: Façade: Virtual storage devices with performance guarantees. In: International Conference on File and Storage Technologies (FAST), San Francisco, CA (2003) 131–144
10. Karlsson, M., Karamanolis, C., Chase, J.: Controllable fair queuing for meeting performance goals. In: IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation (PERFORMANCE), Juan-les-Pins, France (2005) 278–294
11. Chase, J., Anderson, D., Thakar, P., Vahdat, A., Doyle, R.: Managing Energy and Server Resources in Hosting Centres. In: ACM Symposium on Operating Systems Principles (SOSP), Banff, Canada (2001) 103–116
12. Åström, K.J., Wittenmark, B.: Adaptive Control. 2 edn. Electrical Engineering: Control Engineering. Addison-Wesley Publishing Company (1995) ISBN 0-201-55866-1.
13. Bryson, A., Ho, Y.C.: Applied Optimal Control – Optimization, Estimation, and Control. Taylor & Francis (1975) ISBN 0-89116-228-3.
14. Karlsson, M., Zhu, X., Karamanolis, C.: An Adaptive Optimal Controller for Non-Intrusive Performance Differentiation in Computing Services. In: IEEE Conference on Control and Automation (ICCA), Budapest, Hungary (2005)
15. Diao, Y., Hellerstein, J., Parekh, S.: MIMO control of an Apache web server: Modeling and controller design. In: American Control Conference (ACC), Anchorage, AK (2002) 4922–4927
16. Li, B., Nahrstedt, K.: A control theoretical model for quality of service adaptations. In: International Workshop on Quality of Service (IWQoS), Napa, CA (1998) 145–153
17. Lu, C., Abdelzaher, T., Stankovic, J., Son, S.: A feedback control approach for guaranteeing relative delays in web servers. In: IEEE Real Time Technology and Applications Symposium (RTAS), Taipei, Taiwan (2001) 51–62
18. Robertsson, A., Wittenmark, B., Kihl, M., Andersson, M.: Design and Evaluation of Load Control in Web Server Systems. In: American Control Conference (ACC), Boston, MA (2004) 1980–1985
19. Wei, J., Xu, C.Z.: A Self-tuning Fuzzy Control Approach for End-to-End QoS Guarantees in Web Servers. In: International Workshop on Quality of Service (IWQoS), Passau, Germany (2005) 123–135
20. Diao, Y., Lui, X., Froehlich, S., Hellerstein, J., Parekh, S., Sha, L.: On-line response time optimization of an apache web server. In: International Workshop on Quality of Service (IWQoS), Monterey, CA (2003) 461–478
21. Sundaram, V., Shenoy, P.: A practical learning-based approach for dynamic storage bandwidth allocation. In: International Workshop on Quality of Service (IWQoS), Monterey, CA (2003) 479–497
22. Welsh, M., Culler, D.: Adaptive overload control for busy internet servers. In: USENIX Symposium on Internet Technologies and Systems (USITS), Seattle, WA (2003) 43–56
23. Lu, Y., Abdelzaher, T., Lu, C., Tao, G.: 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 (2002) 23–32
24. Wu, K., Lilja, D., Bai, H.: The Applicability of Adaptive Control Theory to QoS Design: Limitations and Solutions. In: International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS), Denver, CO (2005)
25. Liu, X., Zhu, X., Singhal, S., Arlitt, M.: Adaptive entitlement control of resource containers on shared servers. In: IFIP/IEEE International Symposium on Integrated Network Management (IM), Nice, France (2005) 163–176
26. Lu, C., Wang, X., Koutsoukos, X.: End-to-end utilization control in distributed real-time systems. In: International Conference on Distributed Computing Systems (ICDCS), Hachioji, Japan (2004) 456–466