An Observation-based Approach Towards Self-managing Web Servers

Prashant Pradhan, Renu Tewari, Sambit Sahu

Networking Software and Services
IBM T. J. Watson Research Center
Hawthorne, NY 10532
{ppradhan,tewarir, ssahu}@us.ibm.com

Abhishek Chandra*, Prashant Shenoy

Department of Computer Science
University of Massachusetts
Amherst, MA 01003
{abhishek, shenoy}@cs.umass.edu

Abstract

As more business applications have become web enabled, the web server architecture has evolved to provide performance isolation, service differentiation, and QoS guarantees. Various server mechanisms that provide QoS extensions, however, rely on external administrators to set the right parameter values for their desirable performance. Due to the complexity of handling varying workloads and bursty traffic, configuring such parameters optimally becomes a challenge. In this paper we describe an observation-based approach for self-managing web servers that can adapt to changing workloads while maintaining the QoS requirements of different classes. In this approach, the system state is monitored continuously and parameter values of various system resources—primarily the accept queue and the CPU—are adjusted to maintain the system-wide QoS goals. We implement our techniques using the Apache web server and the Linux operating system. We first demonstrate the need to manage different resources in the system depending on the workload characteristics. We then experimentally demonstrate that our observation-based system monitors such workload changes and adjusts the resource parameters of the accept queue and CPU schedulers in order to maintain the QoS requirements of the different classes.

*This research was carried out when Abhishek Chandra was a summer intern at IBM T.J. Watson.

1 Introduction

1.1 Motivation

Current web applications have evolved from simple file browsing to complex tools for commercial transactions, online shopping, information gathering and personalized service. To accommodate this diversity, web servers have evolved into complex software systems with a three-tier architecture consisting of a front-end HTTP server, an application server containing the business logic components, and a back-end database server. The front-end HTTP (web) server in such cases performs a variety of tasks such as (a) dynamic HTML generation, (b) personalized page assembly using scripting languages (e.g., JSP), (c) SSL processing for secure transmission, (d) persistent HTTP protocol processing, to reduce connection setup overheads and improve end-user performance, and (e) communication with the application server components via servlets. In doing so, it interacts in complex ways with the underlying OS mechanisms that manage resources such as the CPU, memory, disk and the network interface. Another emerging trend is the growing popularity of web hosting services that collocate multiple web domains on the the same host machine or a cluster and provide different levels of service to these domains based on various pricing options. In such environments, service differentiation and performance isolation become necessary for efficient operation.

Numerous mechanisms for service differentiation and performance isolation have been proposed in the literature. Such mechanisms for web servers include QoS-aware extensions for admission control[8], SYN policing and request classification[28], accept queue scheduling [2], and CPU scheduling [3]. These mechanisms enable a web server to differentiate between requests from different classes and provide class-specific guarantees on performance (for instance, by providing preferential treatment to users who are purchasing items at an e-commerce site over users who are merely browsing, or by providing better service to institutional investors over individual investors at a financial site). One limitation of these QoS mechanisms is that they rely on an external administrator to correctly configure various parameter values and set policies on a system-wide basis. Doing so not only requires a knowledge of the expected workload but also a good understanding of how various operating system and web server configuration parameters affect the overall performance. Thus, while these QoS mechanisms undoubtedly improve performance, they also *exacerbate the problems of configuration and tuning*—each mechanism provides one or more tunable "knobs" that the system administrator needs to deal with. More importantly, these mechanisms are not independent of one another—depending on the configuration, each mechanism can have repercussions on the behavior of others, which further complicates the configuration process. Furthermore, past studies have made contradictory claims about the utility and benefits of these mechanisms. For instance, one recent study has claimed that the (socket) accept queue is the bottleneck resource in web servers [2], while another has claimed that scheduling of requests on the CPU is the determining factor in web server performance [3]. Thus, it is not evident a priori as to which subset of QoS mechanisms should be employed by a web server and under what operating regions.

The increasing complexity of the web server architecture, the dynamic nature of web workloads [10, 13], and the interactions between various QoS mechanisms makes the task of configuring and tuning modern web servers exceedingly complex. It has been argued that the more complex the system, the greater are the chances of a mis-configuration and sub-optimal performance [4, 9]. To address this problem, in this paper, we develop an adaptive architecture to make web servers *self-managing*. By self-managing, we mean mechanisms to automate the tasks of configuring and tuning the web server so as to maintain the QoS requirements of the different service classes. The emphasis on manageability of computing systems has gained momentum in recent years with the ever increasing complexity of these systems—in fact, several researchers have argued that, in today's environments, the problems of manageability, availability and incremental growth have overshadowed that of the traditional emphasis on performance [12, 18].

1.2 Research Contributions

This paper focuses on the architecture of a self-managing web server that *supports multiple QoS classes*—a scenario where multiple virtual servers run on a single physical server or where certain classes of customers are given preferential service. Assuming such an architecture, we make three key contributions in this paper. (1) We conduct an experimental study using the Apache web server to identify bottleneck resources for different web workloads; our study illustrates how the bottleneck resource can vary depending on the nature of the workload and the operating region. (2) Based on the workloads in our study, we identify a small subset of resource control mechanisms—the incoming request queue scheduler and the CPU share-based scheduler—that are likely to provide the most benefits in countering the performance degradation. (3) We then present an observation-based technique to automate the tasks of configuring and tuning of the parameters of these OS mechanisms. A key feature of this technique is that it can handle multiple OS resources in tandem. Our architecture consists of techniques to monitor the workload and to adapt the server configuration based on the observed workload. The adaptation system can adjust to: (i) a change in the request load, (ii) the QoS requirements of the classes, (iii) the workload behavior and (iv) the system capacity. Since the system dynamically monitors and adjusts the parameters it makes no underlying assumption of the workload characteristics and the parameter behaviors.

We implement our techniques into the Apache web server on the Linux operating system and demonstrate its efficacy using an experimental evaluation. Our results show that we can adjust dynamically to a change in workload, a change in response time goal and a change in the type of workload.

The rest of this paper is structured as follows. Section 2 presents our experimental study to determine the bottlenecks in the Apache request path. Section 3 discusses the architecture and kernel mechanisms used to support multiple classes of web requests. Section 4 presents our framework to configure and tune the web server. Section 5 presents the results of our experimental evaluation. Section 6 discusses related work, and finally, Section 7 present our conclusions.

2 Analyzing the Bottlenecks in Web Request Processing

In this section, we examine the bottlenecks encountered in the processing of web requests. We use Apache as a representative example of a web server and subject it to a variety of different workloads. For each workload, we determine the bottlenecks in the request path at different operating regions. In what follows, we first present a brief overview of the software architecture employed by Apache before presenting our experimental results.

2.1 Architecture of the Apache Web Server

Apache employs a process-based software architecture. Apache spawns a pool of child processes at startup time, all of which listen on a common socket (typically, port 80). A newly arriving request is handed over to one of the children for further processing; the process rejoins the pool after it is done servicing the request and waits for subsequent requests. Apache can vary the size of the process pool dynamically depending on the load—it starts with a certain number of children and spawns additional processes as the load increases. The limit on the maximum number of children is determined by a statically defined parameter, *MaxClients* (this parameter imposes a limit on the number of concurrent Apache processes to prevent memory exhaustion and thrashing in the system). Once this limit is reached, no additional children are created and newly arriving requests must wait for an existing child to become idle before getting serviced. Apache can also terminate child processes when the load decreases, thereby reducing the number of idle processes in the system.

Next, we examine the control path of a web request. Since HTTP employs TCP as its underlying transport protocol, the client first establishes a TCP connection with the server. This is done using a three-way TCP handshake which is initiated by sending a TCP SYN packet to the server. Once the handshake is complete, the new connection is appended to the accept queue of the listening socket. The HTTP request waits in this queue until a child process accepts the connection. In case of HTTP/1.0, each request uses a separate TCP connection, whereas in HTTP/1.1, multiple HTTP requests can share a single connection (the connection is kept open for a timeout duration, during which multiple HTTP requests can be serviced).

For each request, the Apache child process first parses the request, retrieves the requested object (from the disk or the cache) and sends back a response. Dynamic HTTP requests involve additional CPU processing before a response can be generated. Thus, the servicing of each request involves a certain amount of CPU, disk and network I/O.

With this background, we present the results of our experimental study to determine the bottlenecks in the Apache request path.

2.2 Determining Web Server Bottlenecks

The testbed for our experiments consists of an unmodified Apache server running on a Pentium III PC with 512MB RAM and Redhat Linux 7.1. The client workload is generated using an off-the-shelf web workload generator—*httperf* [23]—that can emulate various kinds of workloads (e.g., persistent HTTP, SSL encryption) and different request rates. All machines were interconnected by a 100 Mb/s switched Ethernet and the network was assumed to be lightly loaded in our experiments.

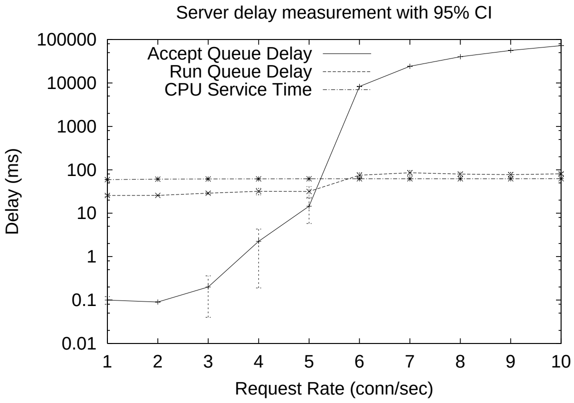
We instrumented the Linux kernel to measure various parameters that affect the performance of web requests, namely (i) the length of the socket accept queue and the time spent by an incoming request in the accept queue, (ii) the amount of CPU time spent in servicing a request, and (iii) the time spent by a request waiting in the CPU run queue. Other metrics such as the network transfer time and the end-to-end response time were measured at the client using httperf. Unless specified otherwise, all kernel and Apache configuration parameters were set to their default values. The only (kernel) parameter that was modified was the maximum length of the accept queue, which was increased from its default value of 128 to 65536 (this was done to avoid TCP SYN packet drops due to accept queue overflow at heavy loads).

For this setup, we examined the performance of Apache for the following workloads: (i) static web requests over non-persistent HTTP connections, (ii) static web requests over persistent HTTP connections, (iii) static requests using SSL encryption, and (iv) dynamic requests using Apache's CGI scripting. Whereas the first two workloads are I/O-intensive, the third is both CPU- and I/O-intensive and the fourth is predominantly CPU-intensive. Due to the memory sizes on our machines, we observed that the OS buffer cache was able to easily cache popular files in memory, and hence, most requests are serviced directly from the cache and did not result in disk I/O. Since most requests are serviced from memory rather than from disk, we find that I/O time is independent of the load and depends only on the file size, and hence, do not report it in our results (this assumption does not hold for scenarios where, for instance, a web request triggers a query in a backend database server; however, such scenarios are outside the scope of this paper, given our focus on web server performance).

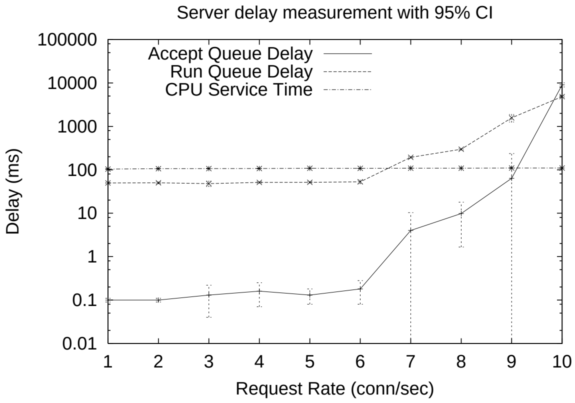
We now present our experimental results. Due to space constraints, we present detailed results only for two scenarios (persistent HTTP and SSL processing).

Static Web Requests using Persistent HTTP

In this experiment, we configured httperf to use persistent HTTP connections and to request multiple (static) files over the same connection. We increased the connection rate and observed its impact on the web server and client performance. As shown in Figure 1(a), at low loads, Apache can easily handle all incoming connections (and requests over those connections); requests do not incur any significant delays in the socket accept queue or the CPU run queue. Note that the persistent nature of each connection causes each Apache process to keep the client connection open for a timeout duration waiting for subsequent requests (which delays its return to the idle process pool). Hence, when the load increases, Apache spawns additional child processes to service newly arriving connections (since existing processes are servicing other connections). As the load increases, the *MaxClient* limit is reached eventually (*MaxClients* was set to 50 in this experiment). Beyond this



(a) Accept Queue bottleneck



(b) Run Queue bottleneck

Figure 1: Bottleneck resources for different workloads

point, the accept queue delay increases rapidly and becomes the dominant factor of the total response time (this is because a newly arriving connection must now wait in the accept queue until an existing child process terminates a persistent connection). Figure 1(a) also shows that the CPU service time and the CPU run queue delay are relatively constant, indicating that most Apache processes are waiting for requests over persistent connections, rather than actively servicing requests. This indicates that the accept queue is the bottleneck resource in this scenario, while the CPU is under-utilized.

Static Web Requests using SSL Encryption

In this experiment, we configure httperf to request static files using SSL encryption over non-persistent HTTP connections. The SSL protocol involves public-key authentication and key exchange during connection setup, after which it uses symmetric key encryption for transmitting the data over the connection. Due to the computational overheads involved in encrypting data, this is a CPU-intensive workload. Like in our previous experiment, we increase the client request rate and measure its impact on server performance. Figure 1(b) depicts our results. The figure shows that the CPU run queue waiting times increase steadily with the load—the larger the CPU load, the greater is the time a request needs to wait in the run queue before it can be scheduled on the CPU (since the CPU is busy servicing other requests). The figure also shows that the CPU run queue delay dominates the server response time. Observe that the CPU *service time* of a request is independent of the load, since the time to service a request (e.g., encrypt data) depends only on the request size. The figure also shows that the accept queue delay is initially small and then increases rapidly beyond a certain load. This is because the CPU saturates at those loads, causing newly arriving requests to wait in the accept queue until an Apache process can be scheduled on the CPU to accept the connection. At very heavy loads, the *MaxClients* limit is reached, further adding to the accept queue delay. Thus, our experiment indicates that the CPU is the primary bottleneck in this scenario. Although the accept queue delays are significant, this is primarily due to the saturation of the CPU, rather than any shortcomings at the accept queue.

We performed two additional experiments that we do not report here due to space constraints. The first experiment involved requests for static web pages over non-persistent connections. Due to the memory sizes on our machine, the OS buffer cache was able to absorb most of the requests, resulting in few disk accesses; consequently, we found the CPU to be the bottleneck resource in this experiment. The second experiment involved dynamic HTML generation using CGI scripting; we found that executing CGI scripts is compute-intensive, causing the CPU to be a bottleneck.

Together, these experiments indicate that depending on the workload and the operating region, different resources can become bottlenecks in the request path. For the workloads that were examined and for our hardware configurations, we observed that the CPU and the accept queue were the primary bottlenecks.[1] This indicates that a web server needs to intelligently detect these scenarios and manage these resources accordingly.

3 Adaptive QoS Architecture

Our experimental study in the previous section highlighted that different resources could become the bottleneck based on the workload characteristics. Based on these insights we choose a small set of kernel mechanisms to control these resources via dynamic resource scheduling. In this paper we target two resources—the accept queue and the CPU—that most affected server performance for our selection of workloads, to highlight the need for multi-resource adaptation. Observe that our goal is not to design new resource control mechanisms; rather it is to pick existing mechanisms in current commercial or open-source operating systems and build an adaptive framework to parameterize and control these mechanisms.

We assume that the web server supports multiple classes of requests (also referred to as *service classes*) each with its specified QoS requirement. In this paper we consider class-specific response time as the default QoS metric. Throughput is another metric that can

[1]Neither the disk nor the network interface became a bottleneck in our experiments. Cache hits in the OS buffer cache prevented the disk from becoming a bottleneck. The network interface did not appear to be a bottleneck either. As noted earlier, these observations may not hold for environments that differ significantly from those considered here—for instance, e-commerce sites with large amounts of backend database I/O.

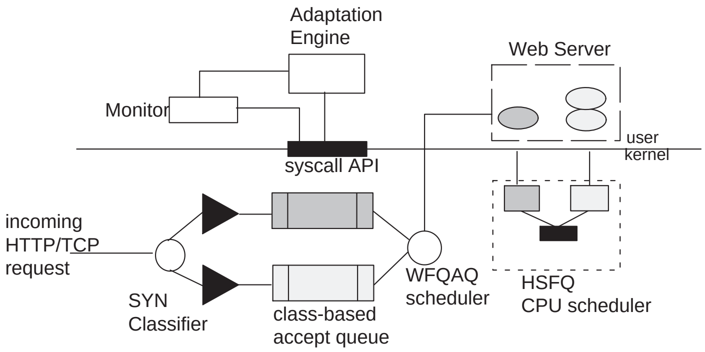


Figure 2: Architecture for Adaptive QoS

be controlled, but discussion of such metrics is beyond the scope of this paper. To control the performance offered to requests within each class, we employ an adaptive QoS architecture that consists of three main components.

- *Kernel resource controllers:* The two resources, the socket's accept queue and the CPU run queue, are controlled by a proportional-share scheduler to meet the performance goals of different service classes. Specifically, we use a weighted fair queuing scheduler for the accept queue, and the hierarchical start-time fair queuing (HSFQ) scheduler for the CPU. A SYN classifier is used to classify incoming TCP connections into their service classes.

- *Monitoring framework :* The monitoring framework continuously obtains measurements from the system for each resource, and each class, which are used by the adaptation engine. Examples of these measurements include per-class delays, request service times and resource utilizations.

- *Adaptation engine:* The adaptation engine uses an observation-based approach to adjust the resource allocations for each class based on the monitored performance and the desired QoS goal. The adaptation progresses on two levels—a local, per-resource level and a global one across resources.

Figure 2 illustrates the interactions between these components. The flow of control during the lifetime of the request is as follows. The kernel performs early demultiplexing and classification of incoming TCP (SYN) packets and assigns each request to a service class. After a request is admitted and added to a class-based accept queue, the weighted fair queuing accept queue (WFQAQ) scheduler determines the order in which the waiting Apache processes accept the requests. After accepting a new request, each Apache process is attached to the corresponding CPU service class of the request and scheduled by the HSFQ CPU scheduler. Through the monitoring framework, the performance of each class is monitored continuously by the adaptation engine. In response to changing workload, the adaptation engine adjusts the shares assigned to each class in the accept queue and the CPU scheduler such that their QoS goals are met.

In what follows, we first describe the kernel mechanisms used in our adaptive QoS architecture and then describe the monitoring framework and the adaptation algorithms.

3.1 SYN Classifier

The SYN classifier uses the network packet headers to perform classification of incoming requests into different service classes. In [28] there is a description of how to extend the classification within the kernel to include application headers. Since a majority of web requests use TCP as the underlying transport, the SYN classifier resides in the TCP/IP processing path. The classifier employs classification rules to determine the class to which an incoming connection belongs. The classifier includes mechanisms for admission control via SYN policing, however, we do not focus on the admission control aspects in this paper. The classification rules, shown in Table 1, are based on the network 4-tuple (IP address and port number). In our prototype on Linux, the *iptables* command is used to insert and delete rules in the kernel packet filtering tables. These filters are maintained by the *netfilter* framework inside the Linux kernel [1].

Filter	QoS Specification
128.1.1.*, 80, *, *	Delay=200ms
128.1.1.*,21, *,*	Delay=1sec
,, 112.3.4.*, *	Thruput=100 req/sec

Table 1: Classification Rules

3.2 Accept Queue Scheduler

For a new incoming request, after the three-way TCP handshake is complete, the connection is moved from the SYN queue (called the partial-listen queue in a BSD-based stack) to the listening socket's accept queue. Instead of a single FIFO accept queue for all requests, our architecture employs a separate accept queue for each service class. Requests in these queues are scheduled using a work-conserving weighted fair queuing (WFQAQ) scheduler. The scheduler controls the order in which requests are accepted from these queues for service by the web server processes. The scheduler allows a weight to be assigned to each class; the rate of requests accepted from a class is proportional to its weight. Thus, the weight setting of a class allows us to control its delay in the accept queue. As soon as an Apache process becomes idle, a request is dequeued from one of the class-specific accept queues in accordance with their weight assignments. Thus, the Apache process pool is *not* statically partitioned across classes. WFQAQ is a work-conserving scheduler—an Apache

Request rate per class	WFQAQ Delay (sec)		
	C1	C2	C3
375,375,375	1.5	1.5	1.5
375,400,400	1.5	1.8	1.9
375, 425, 425	1.5	1.9	2.1

(a) WFQAQ Performance Isolation

WFQAQ shares	Delay (secs.)		
	C1	C2	C3
2:1:1	0.005	2.4	2.2
3:2:1	0.006	1.42	3.15
4:3:1	0.005	0.75	3.68
5:4:1	0.006	0.49	3.89
6:5:1	0.007	0.26	4.15

(b) WFQAQ Delay Control (400 req/sec per class)

Request rate per class	Delay (sec) Static Priority		
	C1	C2	C3
425,425,425	1.4	1.5	1.8
450,425,400	1.5	1.7	1.9

(c) Performance with static priority scheduler

Table 2: Properties of WFQAQ and static priority scheduler

process will not idle if there is a request in any one of the accept queues. If a queue is empty, the unused allocation of that class is proportionately redistributed among other classes.

There are other alternatives for managing the accept queue; we discuss these alternatives briefly and contrast them to our WFQAQ scheduler. The first alternative is to employ a fixed static-priority accept queue [28] that always services a higher priority request before servicing a lower priority request. The problem with a prioritized scheduler is that lower priority classes can be starved by the higher priority classes. As shown in Table 2(c), the lower priority class (C3) gets affected by the request rate of the high priority classes (C1 and C2). In contrast, a proportional-share scheduler like WFQAQ provides performance isolation across classes, since an accept queue buildup in one class does not affect the performance of other classes. If the accept queue of a class becomes full, then further requests are dropped as in the traditional (single FIFO) accept queue scenario. This effect is shown in Table 2(a) that lists the request rate and observed delay for three classes scheduled by a WFQAQ scheduler. Each class is given the same share, i.e., the weight assignments are 1:1:1. Note that even when classes C2 and C3 increase their rates, from 375 req/sec to 400 and 425 req/sec, the delay of class C1 remains unchanged. Classes C2 and C3 eventually start showing errors in the form of client drops, since they only affect their own queue build-up.

The second problem with a prioritized scheduler is that it can only adapt to changing loads in a very coarse-grained manner. Table 2(c) shows that for a given priority ordering among classes, a prioritized scheduler provides exactly one combination of delays that will be seen by the classes based on the offered load of each class. In contrast, a WFQAQ scheduler can offer a wide range of delays for the classes by tuning their share allocation. Thus, the *schedulability region* of a WFQAQ scheduler is larger than that of a prioritized scheduler. Table 2(b) shows the delay values provided by WFQAQ for different weight assignments to 3 service classes, each receiving requests at a rate of 400 requests/sec.

Instead of static priority, another technique is to statically partition the Apache server processes among the service classes and rely on a feedback system to dynamically adjust the number of processes assigned to a class. Although the approach can adapt to changing loads, its primary drawback is that it is non work-conserving—an Apache process assigned to a class can idle if its queue is empty, even though other classes may have pending requests.

A third approach is to employ deadline-based scheduling of tasks (e.g., EDF). This approach assigns a deadline to each request and orders/schedules requests in increasing order of deadlines. An EDF scheduler by itself does not provide performance isolation across classes and additional mechanisms are necessary to limit the utilization of each class.

3.3 CPU scheduler

Traditionally, the CPU scheduler on a Unix-based system schedules all OS application processes using a time-shared priority based scheduler. The scheduling priority depends on the CPU usage of the process, the I/O activity, and the process priority.

To achieve the desired response time goal of a class and provide performance isolation, we use a hierarchical proportional-share scheduler that dynamically partitions the CPU bandwidth among the classes. Specifically, we use the hierarchical start-time fair queuing (HSFQ) [17] scheduler, to share the CPU bandwidth among various classes. HSFQ is a hierarchical CPU scheduler that fairly allocates processor bandwidth to different service classes and uses a class-specific scheduler for processes within a class. The scheduler uses a tree-like structure with each process (or thread) belonging to exactly one leaf node. The internal nodes implement the start-time fair queuing (SFQ) scheduler that allocates weighted fair shares, i.e., the bandwidth allocated to a node is in proportion to its weight. Unused bandwidth is redistributed to other nodes according to their weights. The properties of SFQ, namely: i) it does not require the CPU service time to be known apriori, and ii) it can provide provable guarantees on fairness, delay and throughput received by each process (or thread), make it a desirable proportional-share scheduler for service differentiation.

In our implementation, we use only a 2-level hierarchy (consisting of the root and various service classes). On accepting a web request, each web server process is dynamically attached to the corresponding service class; the CPU share of the class is determined dynamically based on the requirements of the class and the current workload.

One question that arises regarding the CPU control is whether share-based CPU scheduling is required if the number of processes attached to a class can be adjusted dynamically. A direct co-relation between number of processes per class and the CPU bandwidth that a class receives does not always hold. While it may be a valid assumption for small file accesses and single tiered systems, it does not hold in general when processes have different service time requirements, different disk I/O idling times, or are kept alive by HTTP/1.1 for connection re-use across requests. For more fine-grained performance control, a share-based CPU scheduler is required.

3.4 Monitoring Framework

The monitoring framework continuously obtains measurements on the state of each resource and class that are used by the adaptation engine. These measurements can be broadly categorized into per-class, or *local* measurements, and resource-wide, or *global* measurements. Examples of local measurements include per-class delays in a resource, per-class request arrival rates, or the work required by a class's requests in a resource. Examples of global measurements include resource utilization, or global queue lengths.

The monitoring subsystem is essentially a set of kernel mechanisms to extract measurements from each of the resources managed by the adaptation framework. As an example, we briefly describe the per-class delay measurement implemented for the accept queue and the CPU run queue. In case of the accept queue, when a connection is enqueued in the accept queue, we timestamp its arrival in the associated socket data structure. When TCP dequeues a request from the accept queue, as dictated by the accept queue scheduler, we timestamp the departure of the request and compute the time spent in the accept queue. This measurement is aggregated in a running counter together with the number of requests seen by the accept queue. In a similar manner, for CPU, we measure the time spent by a process waiting in the run queue and running on the CPU.

A system call interface is used to allow the adaptation algorithm to perform monitoring as well as resource control. We added an *ioctl* like system call, `sys_multisched()`, to the Linux kernel for this purpose. `sys_multisched()` takes as arguments a command and some command-specific arguments. The commands allow the local class-specific values and global resource values to be queried or updated. For local class-specific measurement, the call arguments identify the command, the resource, the resource-specific metric of interest, and the class identifier. For global measurements they only identify the resource and the metric of interest.

Operationally, two timers are used, *viz* a monitoring timer and an adaptation timer. The values are measured by the monitor every monitoring instant, or "tick", where the time-interval per tick, T_m, is a configurable value. The time-interval between the adaptation instants, $T_a = kT_m$, i.e., an adaptation instant happens after multiple monitoring instants, or every k ticks. The measured values over the k ticks are averaged to give the current value at the start of a new adaptation instant. The value at the previous adaptation instant is exponentially averaged using a weighting factor α. For a resource parameter a, whose exponentially averaged value in the last cycle was a_{prev} and the new set of values at the start of the current adaptation instant were $a_1, a_2, ...a_k$, the new value, a_{cur} is given by

$$a_{cur} = \alpha \cdot a_{prev} + (1 - \alpha) \cdot \frac{\Sigma_{i=1}^{k} a_i}{k}; \quad (0 \leq \alpha \leq 1)$$

4 Adaptation Engine

The adaptation engine builds upon the monitoring, scheduling and control infrastructure described in the previous section. Based on the measured values returned by the monitoring agent, the adaptation algorithm computes and sets new shares and weights in the schedulers in order to meet the QoS goals of each class.

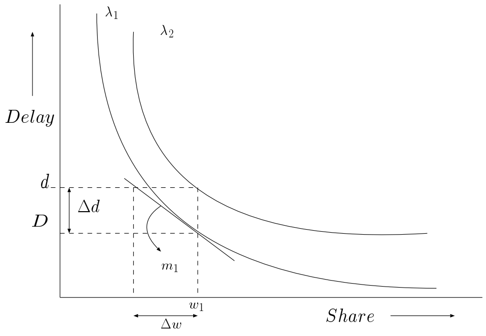


Figure 3: Delay-share relation for different request arrival rates

4.1 Adaptation Techniques

There are three general approaches that can be employed to build an adaptation framework: (i) a control theoretic approach with a feedback element, (ii) an open-loop approach based on a queuing model of the system, and (iii) an observation-based adaptive system that uses run-time measurements to compute the relationship between the resource parameters and the QoS goal.

A control theoretic approach is a powerful technique in general. However, most solutions that apply control theory for web server adaptation require training the system at different operating points to determine the control parameters for a given workload. Moreover, these control parameters have to be re-computed when the workload characteristics change, e.g., from CPU-bound SSL requests to network bandwidth-bound multimedia requests. Secondly, these solutions assume a linear relationship between the resource parameters and the QoS goals for all operating regions. While linear assumptions may hold, in practice, for throughput control they cannot be generalized for other QoS goals such as response time. An assumption that the reciprocal of the response time can be modeled by a linear behavior does not capture the delay relationship correctly. The response time, in general, depends on the utilization of the system and the scheduling policy and is, therefore, difficult to capture by using a linear model throughout the range of system utilization values.

On the other hand, an open loop system, for example, one based on a queuing model, is difficult to solve analytically for complex arrival patterns and service time distributions. Queuing models are useful for steady-state analysis and do not handle transients accurately. Simple approximations of arrival and service time distributions lead to incorrect choice of parameters. Moreover, not all schedulable resources can be modeled as queuing systems.

We chose an observation-based approach for adaptation as it is most suited for handling varying workloads and non-linear behaviors. Figure 3 depicts how delay may vary with share assigned to a class (the share for a class translates to its resource utilization). This figure illustrates that (i) the delay-share relationship may change with the request arrival rate λ_i (as depicted by the two λ curves), and (ii) the delay-share relationship is non-linear even when the request rate remains the same. The basic idea in our observation-based approach is to approximate the non-linear relationship between the delay of a class and its share (or weight), by multiple piece-wise linear parts. The algorithm continuously keeps track of the current operating point of each class on its delay-share curve. The observation-based approach depends on run-time adaptation, and hence is well-suited for highly variable and dynamic workloads.

The observation-based adaptation proceeds on two levels—a local per-resource adaptation and a global system-wide adaptation. The next two sections describe the adaptation algorithm in detail.

4.2 Resource-specific Local Adaptation

The local adaptation algorithm of each resource needs to ensure that each class achieves its QoS (in this case response time) goal for that resource. For each class i, let D_i represent its desired response time and d_i be its observed average delay in that resource. Furthermore, for each class i, the algorithm maintains an estimate of the slope, m_i of its delay-share (or delay-weight) curve at the current operating point. The adaptation algorithm tries to adapt the share of each class, w_i, such that the delay value d_i, lies in the range $[(1-\epsilon)D_i, (1+\epsilon)D_i]$. The adaptation proceeds in the following four steps.

Determining class state: At every adaptation instant, the local adaptation engine computes the current value of d_i from the monitored values, as described in Section 3.4.

At every adaptation instant, the algorithm checks whether each class is missing its local response time goal by comparing the values of d_i and D_i. A class that is missing its goal i.e., $d_i \geq (1+\epsilon)D_i$, is called an *underweight* class. Similarly, a class that is more than meeting its goal, i.e., $d_i \leq (1-\epsilon)D_i$ is called an *overweight* class. Other classes that have their delay within the range of the desired delay are called *balanced classes*. The underweight classes are ordered to determine the most underweight class. The algorithm tries to borrow shares from the overweight classes such that the most underweight class becomes balanced. This redistribution step, however, must ensure that the overweight classes are not over compensated to make them underweight as a result.

Redistribution: For redistributing the share across classes, the algorithm needs to quantify the effect of changing the share allocation of a class on its delay. This is computed by using the slope estimate m_i, at the current operating point on the delay-share curve. The total extra share needed by an underweight class i is given by

$$\Delta w_i = \frac{(d_i - D_i)}{m_i}$$

as shown in Figure 3. The extra share required by the underweight class is not equally distributed among the overweight classes. Instead, the amount of share that an overweight class can donate is based on its sensitivity to a change in share. There are two factors that affect the sensitivity of an overweight class j: (i) its delay slack given by $(D_j - d_j)$, which measures how much better off it is from its desired delay goal, and (ii) the current slope of its delay-share curve m_j, which measures how fast the delay changes with a change in share. Based on these factors, the surplus s_j, for an overweight class j is given by

$$s_j = \frac{(D_j - d_j)}{m_j}$$

The surplus of each overweight class is proportionally donated to reduce the likelihood of an overweight class becoming underweight. The donation, $donation_j$, of an overweight class is a fraction of the required extra share weighted by its surplus, and is given by

$$donation_j = \Delta w_i \cdot (\frac{s_j}{\sum_k s_k})$$

Before committing these donations, we must check that the new delay value does not make the overweight class miss its delay goal. Based on the slope m_j we can predict that the new delay value of the overweight class would be given by

$$d_j{}' = d_j + m_j \cdot donation_j$$

If the new delay value misses the delay goal, i.e., $d_j{}' \geq (1+\epsilon)D_j$, the donation is clamped down to ensure that new delay is within the range of the desired delay. The clamped donation is given by

$$clamped\_donation_j = \frac{[(1-\epsilon) \cdot D_j - d_j]}{m_j}$$

The actual donation of an overweight class is, therefore,

$$actual\_donation_j = min\{donation_j, clamped\_donation_j\}$$

The total donation available to the underweight class i, which is the sum of the actual donations of all the overweight classes, i.e., $\sum_j actual\_donation_j$, is never greater than the required extra share Δw_i.

One underlying principle of the redistribution step is that the overweight classes are never penalized more than required. This is necessary because the slope measurements are accurate only in a localized operating region and could result in a large, but incorrect, surplus estimate. When workloads are changing gradually, it is most likely that the extra share requirements of an underweight class will be small, thereby, making the proportional donation of the overweight classes to be smaller.

Gradual adjustment: Before committing the actual donations to the overweight and underweight classes, the algorithm relies on gradual adjustment to maintain stability. This is another hook to ensure that there are no large donations by the overweight classes. A large donation could change the operating region of the classes which would make the computations based on the current slope value, incorrect.

Hence, we perform gradual adjustment by only committing a fraction β $(0 \leq \beta \leq 1)$, of the computed actual donation, which is given by,

$$commit\_donation_j = \beta \cdot actual\_donation_j$$

The algorithm commits the new shares (or weights) to all the involved classes by using the resource control hooks described in Section 3.4.

Settling: After committing the donations, the adaptation algorithm delays the next adaptation instant, by scaling the adaptation timer, to allow the effect of the changes to settle before making further adaptation decisions. We keep the adaptation cycle short during stable states to increase responsiveness and only increase it when settling is required after a change to increase stability.

The committed donations change the current operating points of the involved classes along their delay-share curves. At the next adaptation instant, the algorithm measures the actual observed change in the per-class delays, and uses these values to obtain updated values of the slope m_i for each class. The updated m_i values are used in the above adaptation equations whenever adaptation is performed next.

4.3 System-Wide Global Adaptation

The system-wide global adaptation algorithm maps the overall response time goals for each class to local response time goals for each resource used by that class. One approach is to use the same value for both system-wide response time goal and the local goal per resource. Although this is a nice choice for initial values, it can reduce performance when different classes have a different bottleneck resource. The main intuition behind our utilization-based heuristic for determining local goals is to give a class a more relaxed goal in its bottleneck resource, i.e., the resource where the class requirements are high relative to the resource capacity.

To determine the per-class resource utilizations the global adaptation engine, at every adaptation instant, uses the monitored values of the work required $C_{i,j}$, by each class i using resource j, and the total capacity C_j of each resource j. While the capacity may be a fixed constant (e.g., MIPS) in the case of CPU, for the accept queue it is the measured process regeneration rate of the web server, i.e., the rate at which connections are accepted from the accept queue.

Let D_i be the global response time goal of class i, and $D_{i,j}$ be the the local response time goal of class i in resource j. The sum of the local response time goals should equal the system-wide goal. The local value depends on the utilization $u_{i,j}$, for the class i in resource j, which is given by,

$$u_{i,j} = \frac{C_{i,j}}{C_j}$$

Using the utilization value, the global response time goal is proportionally allocated between the resources, to give the local response time goals for each class, i.e.,

$$D_{i,j} = D_i \cdot (\frac{u_{i,j}}{\sum_k u_{i,k}})$$

A utilization-based deadline splitting approach has also been used in [16], however, their optimization goal is to balance resource utilization. Our intent, instead, is to examine the workload of each class in isolation and relax the goal in the bottleneck resource for that class.

5 Experimental Evaluation

In this section we evaluate the effectiveness of our system's per-resource and global adaptation algorithms in providing response time guarantees under varying workload conditions. We first demonstrate adaptation of the two system resources—accept queue and CPU—in isolation. We study adaptation behavior for workloads with both deterministic and Poisson request arrival distributions.

Deterministic workloads do not generate significant queuing delays in systems that are not overloaded. With such workloads, the predominant delay is the service time which depends on the resource share assigned to each class. Such workloads are useful to analyze for preemptively-scheduled resources like the CPU, but not for resources like the accept queue where the only delay is caused by queuing. Deterministic workloads allow us to demonstrate the effectiveness of the adaptation algorithm in controlling delays by properly scaling per-class resource shares. On the other hand, Poisson-distributed workloads, which are more representative of real-world workloads, allow us to demonstrate the effectiveness of the algorithm in managing queuing delays. Such delays are relevant for both the CPU and the accept queue resource.

We demonstrate the adaptation behavior of the observation-based approach for: i) changes in workload arrival rates that shift the operating region, ii) changes in response time goals of the classes that can change within a resource based on global system state, and iii) change in workload characteristics that shift the resource bottlenecks.

After evaluating adaptation for each resource along the above dimensions, we evaluate system-wide global adaptation that implements the adaptation machinery for both resources, and adjusts resource allocations in the appropriate resource depending upon the current system workload, current resource utilizations, and the global response time goals.

5.1 Experimental Testbed

The experimental testbed consists of a server machine running a kernel with the adaptation mechanisms and algorithms, and two client machines that generate workload. The server is a 660 MHz P-III machine with 256 MB RAM and runs Linux 2.4.7. Each client machine is a 450 MHz P-II with 128 MB RAM, also running Linux 2.4.7. The machines are connected by a 100 Mbps Ethernet. The server runs Apache 1.3.19 with SSL support enabled. The MaxClients parameter of Apache was set to 150 processes.

The server kernel was modified to implement monitoring, scheduling and control mechanisms for the accept queue and the CPU, as discussed in Section 3. These mechanisms form the building blocks for the adaptation algorithm described in Section 4.

The workload generator used at the clients was httperf [23]. Httperf was chosen because it is an open-loop workload generator that not only allows request rates to be specified as a parameter, but also allows generation of deterministic as well as randomly distributed workloads. To stress different resources in the system we use two kinds of workloads:

- *CGI workload:* In this workload, a CGI script is used that blocks for a variable time duration before returning a response. This models blocking for a back-end database request that reduces the Apache process regeneration rate, thereby, stressing the accept queue without loading the CPU.
- *SSL workload:* The SSL workload models a CPU-intensive workload, which does not stress other resources in the system for moderate request rates.

In the experiments that follow, the monitoring framework records the measurements every system "tick" whose value is set to be 5 seconds. For deterministic workloads, adaptation is triggered every 10 ticks in the stable state. In case of Poisson workloads, where the delays show significantly more deviation about their mean, adaptation is triggered every 40 ticks to avoid over-reaction to transient delays. To allow the system to settle after a share is changed, the adaptation interval is increased by a factor of 2.

5.2 CPU Adaptation

For evaluating the adaptation behavior of the CPU, we choose SSL requests as the CPU-intensive workload. The clients request an SSL-encrypted file from the server at a given rate. At the server, response time goals are specified for two classes. In each experiment, we start with an equal share allocation to each class.

Figure 4 illustrates the results of CPU share adaptation with a varying workload request rate and a deterministic arrival distribution. The CPU target delay for both classes was 0.1 seconds. Clients of both classes generate a combined aggregate workload of 12 SSL requests/sec. The fraction of the requests coming from each client was varied from 1:1 to 1:2 to 1:1 to 2:1, with the transitions occurring at 100 ticks, 500 ticks, and 900 ticks respectively. In other words, the class pair had request rates of (6 req/sec, 6req/sec) from 0 to 100 ticks, (4 req/sec, 8 req/sec) from 100 to 500 ticks, (6 req/sec , 6 req/sec) from 500 to 900 ticks, and (8 req/sec, 4 req/sec) from 900 to 1200 ticks. Figure 4(a) is a plot of the average per-class delays with time, and shows that adaptation was successfully triggered in each case such that the

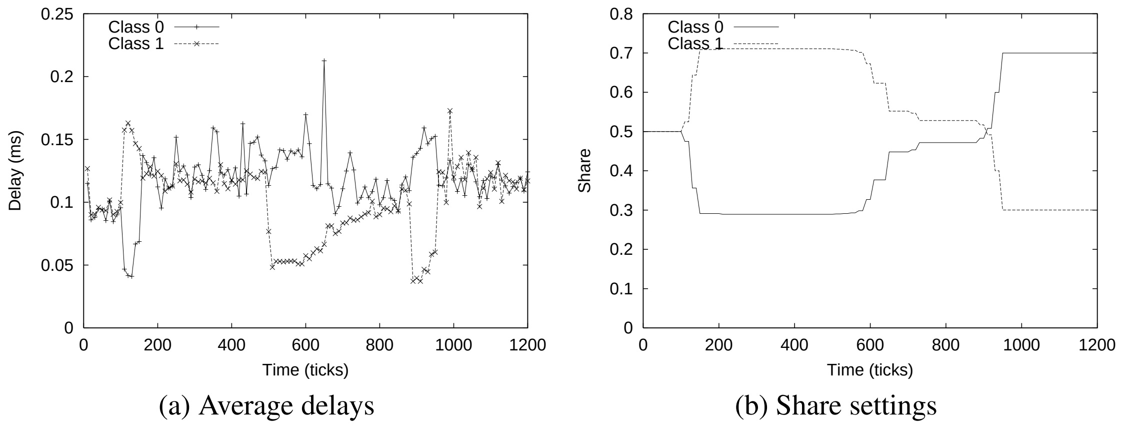
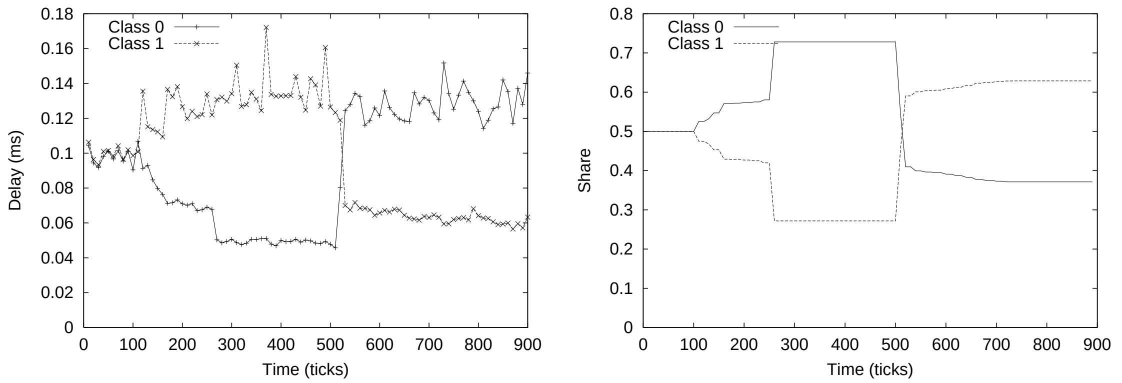


Figure 4: CPU Adaptation for deterministic request arrivals and dynamically changing request rates.

response time of each class was close to its goal. Figure 4(b) plots the relative shares assigned to each class. As the figure shows, share of class 1 was increased at the first transition to handle its increased load. This share could be borrowed from class 0 because it had a reduced load. When the request rates were balanced again at the second transition, share of class 0 was increased to rebalance the previous share setting. Finally, to handle the increased load of class 0, its share was increased at the expense of class 1. Thus, the figure demonstrates the gradual share adaptation being performed by the algorithm in the CPU scheduler.

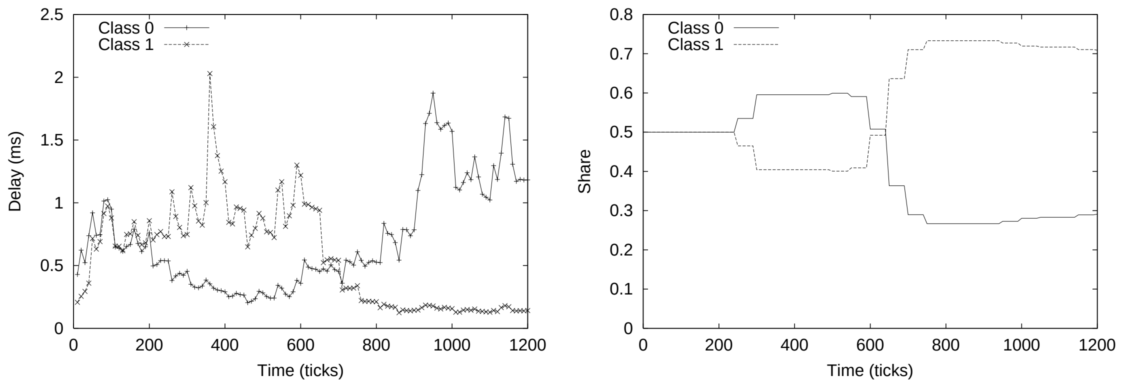
Figure 5 illustrates the results of CPU share adaptation with varying response time goals and a deterministic arrival distribution. Both clients send requests at the rate of 6 SSL requests/sec. Initially the goals of both classes were set to be equal. After 100 ticks, the response time goal of class 0 and 1 was changed to 0.05 seconds and 0.15 seconds respectively. After 500 ticks, the response time goal for these classes was reversed. Note that this reversal causes a large relative change in the response time goals. We use this to stress the adaptation algorithm and verify that large changes do not send the system into oscillations. Figure 5(a) plots the average per-class delays and demonstrates the adaptation to the changes, whereas 5(b) shows the CPU share adjustments performed by the adaptation algorithm.

The above experiments used a deterministic workload to show that the adaptation algorithm can adjust shares to handle changes in request rates and target delays. Next, we study the effectiveness of the adaptation algorithm in managing queuing delays. This is done by using a workload with Poisson request arrivals. Both clients generate requests whose arrival is Poisson distributed with mean 6 requests/sec. During the first 200 ticks, the queue length is allowed to settle, and adaptation does not trigger during this period. Then, at 200 ticks, class 0 is given a goal of 0.25 seconds, whereas class 1 is given a goal of 1 second. At 600 ticks, these goals are reversed, which is again a large relative change. Figure 6 shows the adaptation results. Figure 6(a) plots the *average* delays that are seen by the adaptation algorithm while making adaptation decisions. The weight adjustments made by the algorithm are shown in figure 6(b). Note that the share adjustment done by the algorithm at the second transition is larger and faster than that done at the first transition. The reason is that at the second transition, the lower delay class 0 has more slack in terms of donating from its share to class 1.



(a) Average delays (b) Share settings

Figure 5: CPU Adaptation for deterministic request arrivals and dynamically changing response time goals.



(a) Average delays (b) Share settings

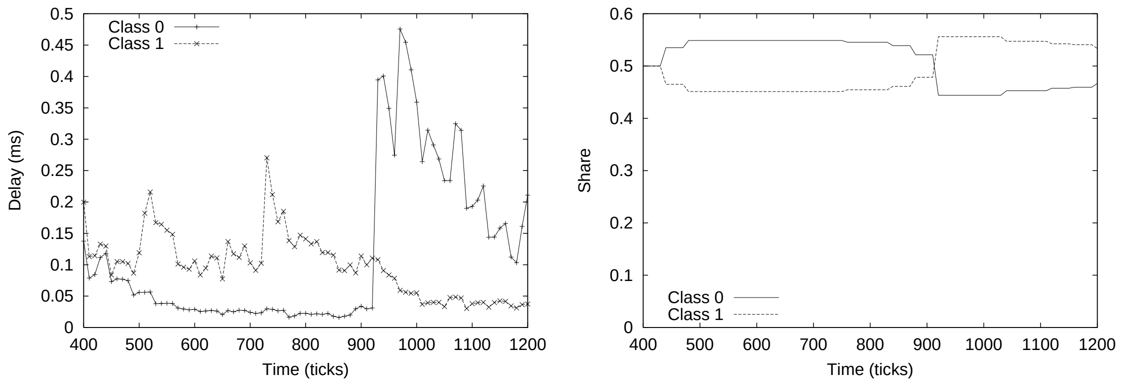
Figure 6: CPU Adaptation for Poisson request arrivals and dynamically changing response time goals.

5.3 Accept Queue Adaptation

For evaluating accept queue adaptation behavior, we use the CGI workload as described earlier. Again, in each experiment we start with an equal share allocation to each class. Note that since the only kind of delay in the accept queue is the queuing delay, only workloads with Poisson-distributed arrivals are relevant. Figure 7 shows the accept-queue share adaptation for varying response time goals. Both classes of clients generate requests whose arrival is Poisson distributed with a mean of 24.6 requests/sec. During the the first 400 ticks, the queue length is allowed to settle. During this period, the response time goal is kept at a high value for both classes, so that adaptation does not trigger. Then, at 400 ticks, class 0 is given a goal of 0.05 seconds, whereas class 1 is given a goal of 0.15 seconds. At 900 ticks, a large relative change is made by reversing these goals. Figure 7(a) plots the average per-class delays and figure 7(b) shows the accept queue share adjustments. As can be seen from the graphs, the adaptation algorithm changes the shares for the classes to meet their delay goals. We do not show the initial 400 ticks of the experiment, as there is no adaptation taking place there.

5.4 System-Wide Adaptation

In this experiment we demonstrate the combined adaptation of both resources when a change in the type of



(a) Average delays

(b) Weight settings

Figure 7: Accept queue adaptation for Poisson request arrivals and dynamically changing response time goals.

workload shifts the bottleneck resource.

For the experiment shown in Figure 8, the clients alternate between generating CGI and SSL workloads. To keep the delay values in each resource comparable, we use a combination of an SSL workload with deterministic arrivals and a CGI workload with Poisson arrivals. Figures 8(a) and (b) plot the average CPU delay and the average accept queue delay respectively, for each class.

The experiment proceeds in three phases.
From 0 to 400 ticks, the clients generate SSL requests at the rate of 6 requests/sec. No adaptation is triggered for the first 100 ticks to allow the system state to stabilize. At 100 ticks, the *global* response time goal of class 0 is set to 0.05 seconds and that of class 1 is set to 0.15 seconds. For the rest of the experiment these global target delays are kept fixed. As seen in these figures, the accept queue delay is negligible (around 0.002 secs) for the first 400 ticks since the workload is CPU-intensive. Hence, the entire delay budget is available to the CPU. As the graph shows, the CPU shares adapt to provide each class with their target delay values.

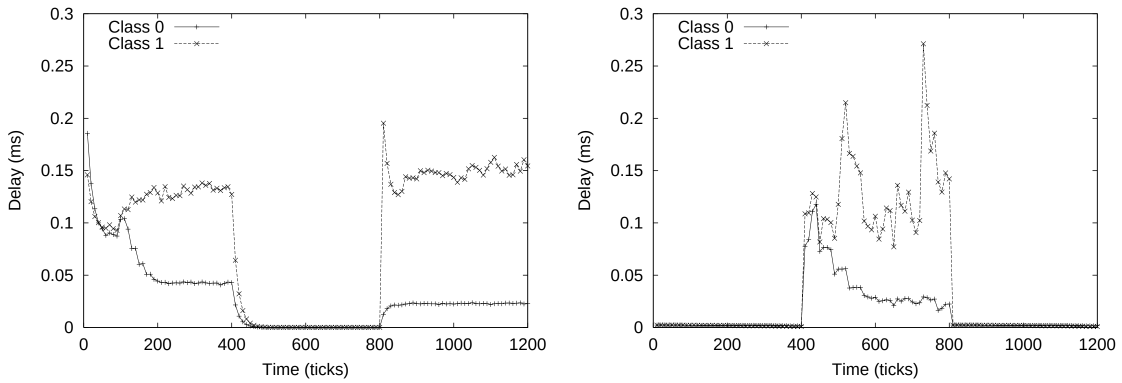
Between 400 to 800 ticks, the clients switch from an SSL workload to a CGI workload with a request rate of 24.6 requests/sec each. This reduces the CPU delay to a negligible value (around 0.0002 seconds) but ramps up the accept queue delay. Most of the delay budget for each class is now available for the accept queue. The accept queue adaptation algorithm responds by adjusting shares to achieve the target delays.

Finally, from 800 to 1200 ticks, the clients switch back to an SSL workload, thus making the CPU the bottleneck resource again. Moreover, the request rates of the clients are also changed to 4 requests/sec and 8 requests/sec respectively. Once again, as shown in the graphs, the accept queue delay becomes negligible while the CPU scheduler parameters are adjusted to help the classes achieve their goal.

This experiment demonstrates the ability of the system to choose the appropriate resource to adapt with changes in the type of workload, and to trigger the appropriate local adaptation to meet per-class response time goals.

6 Related Work

Several approaches for self-managing systems have been proposed in the literature in the context of storage systems [6, 22, 27], general operating systems [25], network services [11], etc. Our focus is to design adaptive techniques to make web servers self-managing while providing QoS guarantees to various customer classes.



(a) Average delays for CPU (b) Average delays for accept queue

Figure 8: Delays for system-wide adaptation for the CPU and the accept queue.

Recently, several research efforts have focused on the design of adaptive web servers. A control theoretic approach for adaptation has been proposed in [2, 21, 30]. This approach involves a training phase using a given workload to perform system identification, based on which a controller is designed that assumes a linear relationship between the QoS metric and the scheduler parameters. Unlike this effort, we employ an alternate observation-based approach for adaptation. Since delay is not linearly related to the share parameters of proportional-share schedulers, and the system model changes with variations in the workload, we perform adaptation by measuring the system state on a continual basis and adapting based on the current operating region. Thus, system identification is an ongoing process in our system, and while we assume linearity around a particular operating point, the operating region as a whole is assumed to be non-linear.

A number of recent and ongoing research efforts have looked at various aspects of providing QoS support for web servers. WebQoS [8] is a middle-ware layer that provides admission control and service differentiation in user space. Unlike the webQoS effort, the focus of our work is not to design new scheduling or resource management mechanisms per se, rather it is to to design an adaptive framework to effectively parameterize existing mechanisms. An adaptive mechanism for admission control for web Servers is described in [19]. Goal-based CPU scheduling using coarse-grained resource allocation techniques for meeting service level agreements has been studied in the context of WLM [5]. In contrast, our work focuses on the combined fine-grained adaptation of multiple resource allocations both in terms of resource units as well as the time scale.

To achieve performance guarantees on a web server, several research efforts have developed predictable resource management mechanisms and techniques for the host operating system. Resource Containers [7] is a kernel mechanism for accurate accounting of resource usage that can be used for service differentiation on a web server. SFQ [17], BVT [15], SMART [24] are predictable scheduling algorithms that can be employed as basic scheduling mechanisms in the kernel. Kernel mechanisms for early classification and managing of accept queue delay have been proposed in [14, 28]. Our work is complementary to the development of such mechanisms. In fact, we assume the existence of such mechanisms and show how to automate the task of parameterizing these mechanisms to achieve self-manageability in the system.

Previous work on resource management for web servers has typically focused on individual resources. [3] has proposed a CPU scheduling algorithm to dynamically distribute CPU bandwidth to Apache processes. Connection setup delay has been identified as the bottleneck in [21] that proposes a scheduling scheme to manage the accept queue. A key contribution of our work was that we showed the need for managing multiple resources, and developed an adaptation technique for controlling multiple resources dynamically.

Many research efforts have looked at resource control from the application perspective. [20] proposes a mechanism to guarantee end-to-end delays for a periodic real-time application with well-defined stages. SEDA [29] is a framework for designing applications that allows controlled resource allocation for each application stage. In [26], a feedback-driven scheme has been proposed that uses application-specific indicators to determine the resource allocation. Most of these approaches require knowledge or make assumptions about the application structure. In our work, we try to infer the application behavior through system-level observations and don't require any knowledge of the application internals.

7 Conclusions and Future Work

In this paper, we proposed an observation-based approach for self-managing web servers that can adapt to changing workloads while maintaining the QoS requirements of different classes. First, we illustrated the need to manage different resources for different kinds of workloads. Later, we described an adaptation framework which monitors the system state continuously and adjusts the various resource parameters to maintain the response time requirements of different classes.

As part of an ongoing effort, we are extending the scope of the adaptation architecture to include other system resources such as disk arrays, network interfaces, etc. This includes integrating the adaptation system with the admission controller. In future we plan to investigate more varieties of web workloads and server architectures, in particular, workloads that involve accessing a back-end server and multi-tier server architectures that include a web server, an application server and a back-end database. We would also like to explore the possibilities of using our adaptation technique in other self-managing scenarios such as large storage systems, database systems, etc.

Overall, we believe that an observation-based approach is a useful technique to adapt to unpredictable loads and other system factors, and our techniques show how this approach can be applied in a web server environment.

References

[1] Netfilter: Firewalling, NAT and packet mangling for Linux 2.4. http://netfilter.samba.org, 2002.

[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*, 13(1), January 2001.

[3] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing Differentiated Quality-of-Service in Web Hosting Services. In *Proceedings of the SIGMETRICS Workshop on Internet Server Performance*, 1998.

[4] G. Alvarez, K. Keeton, A. Merchant, E. Riedel, and J. Wilkes. Storage Systems Management. Tutorial presented at ACM Sigmetrics, June 2000.

[5] J. Aman, C.K. Eilert, D. Emmes, P Yocom, and D. Dillenberger. Adaptive Algorithms for Managing a Distributed Data Processing Workload. *IBM Sytems Journal*, 36(2):242–283, 1997.

[6] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles around Storage Administration. In *Proceedings of the Conference on File and Storage Technologies*, January 2002.

[7] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, February 1999.

[8] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, 13(5), September 1999.

[9] E. Borowsky, R. Golding, A. Merchant, E. Shriver, M. Spasojevic, and J. Wilkes. Eliminating Storage Headaches Through Self-Management. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, October 1996.

[10] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of Infocom'99*, March 1999.

[11] A. Brown, D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiatowicz, and D. Patterson. ISTORE: Introspective Storage for Data-Intensive Network Services. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, March 1999.

[12] A. Brown and D. Patterson. Towards Maintainability, Availability, and Growth Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[13] M R. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[14] P. Druschel and G. Banga. Lazy receiver processing (LRP): a network subsystem architecture for server systems. In *Proc. of OSDI*, pages 91–105, October 1996.

[15] K. Duda and D. Cheriton. Borrowed Virtual Time (BVT) Scheduling: Supporting Lantency-sensitive Threads in a General-Purpose Scheduler. In *Proceedings of the ACM Symposium on Operating Systems Principles*, December 1999.

[16] K. Gopalan and T. Chiueh. Multi-resource Allocation and Scheduling with Real-time Constraints. In *Proceedings of MMCN*, January 2002.

[17] P. Goyal, X. Guo, and H. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, October 1996.

[18] J. Hennessy. The Future of Systems Research. *IEEE Computer*, pages 27–33, August 1999.

[19] H. Jamjoom and J. Reumann. QGuard:Protecting Internet Servers from Overload. Technical report, University of Michigan, CSE-TR-427-00, 2000.

[20] K. Jeffay. On Latency Management in Time-Shared Operating Systems. In *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, May 1994.

[21] C. Lu, T. Abdelzaher, J. Stankovic, and S. Son. Feedback Control Scheduling in Distributed Systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, June 2001.

[22] C. Lu, G. Alvarez, and J. Wilkes. Aqueduct: Online Data Migration with Performance Guarantees. In *Proceedings of the Conference on File and Storage Technologies*, January 2002.

[23] D. Mosberger and T. Jin. httperf – A Tool for Measuring Web Server Performance. In *Proceedings of the SIGMETRICS Workshop on Internet Server Performance*, June 1998.

[24] J. Nieh and M S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the ACM Symposium on Operating Systems Principles*, December 1997.

[25] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 1997.

[26] D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, February 1999.

[27] V. Sundaram and P. Shenoy. Bandwidth Allocation in a Self-Managing Multimedia File Server. In *Proceedings of the Ninth ACM Conference on Multimedia*, October 2001.

[28] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *Proceedings of the Usenix Annual Technical Conference*, June 2001.

[29] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2001.

[30] R. Zhong, C. Lu, T. F. Abdelzaher, and J. A. Stankovic. ControlWare: A Middleware Architecture for Feedback Control of Software Performance. In *Proceedings of ICDCS*, July 2002.