A Resource Management Framework for Predictable Quality of Service in Web Servers

Mohit Aron Sitaram Iyer Peter Druschel

Department of Computer Science
Rice University

DRAFT—DO NOT DISTRIBUTE

Abstract

This paper presents a resource management framework for providing predictable quality of service in Web servers. The framework allows Web server and proxy operators to ensure a minimal quality of service, expressed as an average request rate or average response time, for a certain class of requests (called a *service*), irrespective of the load imposed by other requests. A measurement-based admission control framework determines whether a service can be hosted on a given server or proxy, based on the measured statistics of the resource consumptions and the desired QoS levels of all the co-located services. In addition, we present a feedback-based resource scheduling framework that ensures that QoS levels are maintained among admitted, co-located services. Experimental results obtained with a prototype implementation of our framework on trace-based workloads show its effectiveness in providing desired QoS levels with high confidence, while achieving high average utilization of the hardware.

1 Introduction

As the World Wide Web experiences increasing commercial and mission-critical use, users and content providers expect high availability and predictable performance. Towards this end, work is being done in scalable Web infrastructure [15, 2], Web content caching [21, 12], provision of differentiated services in the Internet [23, 17], and differentiated services in Web servers and proxies [4, 6, 19, 10, 8].

This paper focuses on the latter of these problems, namely providing predictable and differentiated services in Web servers and proxies. Web site and proxy operators often wish to ensure that requests for certain Web pages or requests from certain clients receive a minimal level of quality of service, regardless of the load imposed by other requests. We present a feedback-based resource scheduling framework that is able to ensure a specified QoS level (e.g., average throughput or average response time) for a class of requests, despite varying resource demands (CPU, memory, disk and network bandwidth) and competing requests, as long as the collective resource requirements do not exceed the available resources in the system.

A closely related problem arises in Web hosting sites and content distribution networks [1] who wish to provide predictable QoS levels to multiple *virtual sites* hosted on the same system. The problem for the hosting server or proxy operator is to co-locate virtual sites in such a way that (1) the contracted QoS levels are maintained with high probability, and (2) the average hardware utilization is high. To address this problem, we present a novel framework for measurement-based admission control of Web services that allows proxy and Web hosting operators to decide if a given service (virtual site) can be co-located with other sites on the same machine or cluster (capacity planning), while maintaining service contracts for each site and achieving high hardware utilization.

With our approach, a new service with unknown load and resource demand characteristics is first operated in "trial" (best effort) mode, and on a server node with plenty of available resources. During this phase, the site is exposed to its regular live workload, and both its load characteristics (average throughput and response time) as well as its demand for various resources (CPU, memory, disk and network bandwidth) are recorded by the system. Using this data, the system calculates a statistical envelope of load as well as demand for server resources over a period of time.

Based on this statistical envelope and using our framework, the hosting server or proxy operator can predict the resource demands (and thus the cost) needed to guarantee a certain QoS level, and can offer the content provider an appropriate contract (service level agreement). Once the content provider has contracted for a given QoS level (which

can vary based on time of day, day of week, etc.), the hosting server/proxy operator can use the service's measured statistical envelope to decide which services can be co-located on a given server node or cluster.

Once a set of services have been co-located on a given server node, our resource scheduling framework guarantees that (1) all service contracts are satisfied, as long as the total resource requirements do not exceed the capacity of the hardware, and (2) unused resources are allocated to services whose current load exceeds the contracted service rate. All the while, load and resource demands of each service are continually monitored to detect changes in a site's workload and resource demands, which may require relocation of the service or a change in contract.

In this paper, we describe the design and implementation of our measurement-based admission control and scheduling framework. We present results of a performance evaluation based on synthetic and trace-based workloads including both static and dynamic content requests. The results suggest that our framework is able to predict which sites can be co-located with high confidence and that it can satisfy service contracts while maintaining high resource utilization.

The rest of this paper is organized as follows. Section 2 presents some background on resource management and quality of service in Web servers. The design of our framework is presented in Section 3, and our prototype implementation is briefly described in Section 4. Section 5 presents experimental results obtained with the prototype. Related work is covered in Section 6 and Section 7 offers some conclusions.

2 Background

Providing predictable and differentiated quality of service to Web users on an end-to-end basis requires appropriate resource management in the Internet, in the Web server or proxy that provides the requested resource, and in some cases (e.g., when using streaming media) in the client's browser and operating system. In this paper, we focus on the provision of predictable services in the Web server or proxy. Solving the problem in the server/proxy is an important part of the overall solution, because user-perceived Web performance (i.e., response time) is increasingly dominated by server delays, especially when contacting busy servers [7].

The framework presented in this paper builds on prior work in mechanisms for performance isolation in operating systems [18, 11, 24, 6, 10] and in proportional share scheduling policies for CPU and disk bandwidth [16, 9]. Mechanisms like Resource Containers [6] allow fine-grained resource accounting and scheduling for individual services in a Web server. When combined with proportional resource schedulers, they enable the provision of *differentiated* services, where a predetermined fraction of the server's resources is guaranteed to be available to each service. This paper extends this prior work by presenting a complete framework for the provision of *predictable services*, that is, the ability to meet a predetermined average throughput or average response time for each service.

For the purposes of this paper, we define "provision of predictable quality of service" as the ability of a Web server or proxy to guarantee, with high probability, that the server is able to either (1) handle a given minimal average request rate, or (2) ensure a given maximal average response time, for a given set of URLs (a virtual site) and/or a given client community during a certain period of time. Request rate and average response time are measured here as averages over an interval that is much larger than the time it takes to handle a single request. A related problem in Web server QoS is to ensure a maximal response time or a minimal data rate for *individual* Web requests. We do not consider the latter problem in this paper.

3 A framework for measurement-based QoS

In this section, we present our framework for providing measurement-based quality of service in server systems. The framework consists of (i) an *admission control framework* for admitting services into the system and determining their contracts, and (ii) a *resource scheduling framework* that ensures contracts are met despite varying resource demands.

We begin by defining some terminology. A *system* consists of the hardware and software resources necessary for hosting services. A *service* defines a set of requests, such that the resources used in serving the requests are accounted for and scheduled separately from resources used to serve requests in different services. Thus, a service constitutes a *resource principal* in the system. The set of requests belonging to a service is typically defined by the requested URL and/or or the identity of the client issuing the request.

CPU time, memory pages, disk and network bandwidths each define a *resource class*. An *operator* operates systems (servers and proxies) that are used for hosting services on behalf of one or more *content providers*. The content provider provides the software and data necessary to process requests to a service.

A *QoS metric* is a service specific metric that measures the service's performance and hence the quality of service it is providing to its users. Example QoS metrics are average number of requests per second or average response time. A *contractual target* is a specific value of the QoS metric agreed upon between the operator and the content provider.

The contract between operator and content provider is said to be in *violation* if, despite sufficient load on the service, the system is unable to provide sufficient resources to the service in order for it to meet its contractual target.

The *service type* determines the strength of a contract. For *guaranteed services*, a system must always provide sufficient resources to a service in order to meet its contract. *Predictive services* allow a weaker contract where probabilistic contract violations are acceptable. *Best-effort services* do not have contractual targets and are only allocated remaining resources once the contractual targets of guaranteed and predictive services are met.

The rest of this section describes the admission control framework, the resource monitor, and the feedback-based scheduling framework which form the basis of our approach for providing capacity planning and predictable quality of service.

3.1 Admission Control Framework

Prior to settling on a contract for a service, the operator must determine the resource needs required in order to meet its contractual target. The goal is to ensure that the system remains capable of meeting the contractual targets of guaranteed services, while those of predictive services should still be met with sufficiently high probability. Services of type best-effort do not need to go through the admission control phase; the resource scheduling framework ensures that the contractual targets for guaranteed and predictive services are met before any resources are given to best-effort services.

The problem of determining resource needs is complicated by (i) varying client load (measured in requests per second) on the service over time, and (ii) varying resource demands to sustain a given load. For example, the load during a 24-hour period in our trace from IBM's main Web site (www.ibm.com) varies by a factor of five. Resource demands can vary for a given load when, for example, the average size of the requested content differs at various times, when changing request locality causes different amounts of disk traffic per request, or when different amounts of processing are required for generating dynamic content.

Before admitting a service into the system, the service goes through a *trial* phase where its resource requirements are determined. For this purpose, the service is hosted on a trial system, which has plenty of uncommitted resources. (The trial system may be a special server node set aside for this purpose, or it may be a normal *live* system that hosts other services under contract). The trial system is assumed to have the same hardware characteristics as the system in which the service is to be admitted. The resource usages and the QoS metric (e.g., throughput or response time) for the service are monitored for a period of time that is deemed long enough to determine the service's resource needs with high confidence. The required contractual target can either be already specified by the content provider, or the operator can determine the feasible targets that can be supported.

The load on many Web sites tends to follow a periodic pattern, with a fixed period T (e.g., 24 hours). Let t denote one of k discrete, fixed length intervals in the period T. The length of the interval and the period are tunable depending upon the characteristics of the client load. We describe the average resource need during the interval t of service i for resource class r by a family of random variables $U_{ir}(t)$.

The QoS metric can exceed the contractual target when there are available resources in the system after the contractual targets of all services are met. However, the recorded resource need does not include excess resources consumed when the QoS metric exceeds the contractual target.

The random variables $U_{ir}(t)$ are continuously sampled based on usage reports by the resource scheduling framework. For this purpose, samples are collected for each interval. Let $usage_{ir}(t)$ denote the resource usage for service i for resource class r during interval t expressed as a percentage of the total resources in class r. Let $qos_i(t)$ denote the QoS metric during interval t and let $target_i$ denote the contractual target for service i. Without loss of generality, we assume higher values of $qos_i(t)$ to be more desirable[1]. Then, at any time interval, the value of the random variable $U_{ir}(t)$ is sampled as follows:

$$U_{ir}(t) = \begin{cases} usage_{ir}(t) & qos_i(t) \le target_i \\ usage_{ir}(t) * target_i / qos_i(t) & otherwise \end{cases}$$

Intuitively, the above estimates the resource need by linearly scaling the reported usage when the value of the QoS metric exceeds the contractual target. Note that our framework makes the simplifying assumption that $qos_i(t)$ and resource consumption are approximately linearly related.

Our results indicate that this linearity assumption is reasonable for CPU time, disk and network bandwidth, but not for the main memory resource. The amount of memory required for the service to operate is determined by increasing the memory allocated to the service in a step-wise fashion until a knee is observed in the disk utilization when plotted as a function of the allocated memory. An equivalent amount of main memory is then reserved for the service when it is admitted into the live system. If no knee is observed before allocating all the available memory, the default amount of memory allocated at the start of admission control is reserved.

[1] For QoS metrics such as response time, $qos_i(t)$ can be computed by reciprocating the value of the QoS metric

Estimating the disk resource needs during the trial phase also requires some care. Experimental results indicate that the combined disk needs of a set of services may exceed the sum of their individual disk needs, as measured by running them stand-alone on an idle system. This is because the interleaving of disk requests from different services increases the disk seek times and rotational latency overheads. To account for this, our disk scheduler implements a feature that does not permit trial services to occupy disk resources continuously for more than 50 milliseconds. If disk requests from other services under contract are not pending after 50 milliseconds of disk usage by the trial service, a disk request for a random disk block is injected into the disk queue. This approximates the likely interference from other services when the trial service is admitted into the live system. Experimental results show that disk needs measured for the trial service using this method reflect those observed when hosted on the live system.

Let P be the set of predictive services and G be the set of guaranteed services currently admitted in the system. The admission control decision is made as follows:

Predictive service: Admit the trial service S if for all intervals t the following holds:

$$\forall_r \; P\Big(\sum_{i \in P \cup G \cup \{S\}} U_{ir}(t) \; \leq \; thresh \Big) \; \geq \; C \tag{1}$$

That is, the probability that the sum of the resource needs during any interval t over all admitted services plus the trial service sums to less than a threshold $thresh$ must remain high (higher than a confidence value C where $0 < C < 1$).

The above equation ensures that the contracts of all predictive services shall be met with high probability after the service under trial is admitted into the system.

Guaranteed service: Admit the trial service S if the following equation holds for all intervals t in addition to equation 1.

$$\forall_r \; \max_t \; \hat{U}_{Sr}(t) \; + \; R_r \; \leq \; 100 \tag{2}$$

where $\hat{U}_{ir}(t)$ is the maximal observed resource need of service i for resource class r during interval t, and

$$R_r \; = \sum_{i \in G} \; \max_t \; \hat{U}_{ir}(t) \tag{3}$$

Equation 2 ensures that the system has sufficient resources to cover the worst case resource needs of S. Equation 1 ensures that contracts for existing predictive services are met with high probability after S is admitted.

If the above conditions are satisfied, then resources corresponding to $\max_t \; \hat{U}_{Sr}(t)$ are reserved for every resource class r and the service is admitted into the live system under contract.

The threshold $thresh$ used in equation 1 is a fraction of the total resources available in the system, and is intended to provide a *safety margin* of available resources. It allows the resource scheduling framework to (i) to discover and measure minor increases in resource needs of any service under contract, and (ii) to reduce contract violations of predictive services. The choice of the value of $thresh$ involves a tradeoff; a high value is desirable for attaining high utilization, while a low value is desirable for providing a large safety margin. Experimental results with our prototype indicate that a value of 90% for $thresh$ works well in practice.

The probability in equation 1 is estimated by approximating the probability distribution of the random variable $U_r(t) \; = \; \sum_i \; U_{ir}(t)$ by the distribution of the observed samples of $U_r(t)$. For example, the probability $P(U_r(t) \leq 90\%)$ is estimated by computing the fraction of samples that are all less than 90%.

3.2 Resource Monitor

The resource requirements of services hosted on a live system can change over time. This can result from either changing patterns of load on the services, or from changing resource requirements to maintain the same contractual target. The former can cause contract violations only for predictive services while the latter can cause contract violations even for guaranteed services. The resource monitor serves to warn the operator to take corrective action when the system is in danger of violating contracts. Corrective actions may consist of adding more resources to the system (e.g., by upgrading the hardware or by adding more nodes to a cluster), moving some services to other servers or proxies, or renegotiating the contracts of some services.

The resource monitor detects when the system is in danger of violating contracts by checking equation 1 for every time interval t in the period T. That is:

$$\forall_r \; P\Big(\sum_{i \in P \cup G} U_{ir}(t) \; \leq \; thresh\Big) \; \geq \; C \tag{4}$$

The probability in equation 4 is estimated from the relative frequency[2] of the event when the sum total of the resource needs are less than $thresh$. If during any interval t, equation 4 is violated, the system is considered to be

[2] An event A that occurs n_A times in n repetitions of an experiment is said to have a relative frequency of n_A/n.

in danger of violating contracts. The operator of the system can choose to take corrective action after one or more violations of equation 4.

Our resource scheduling framework trusts the services hosted on the system to report the correct value of the QoS metric. This is reasonable in environments such as Web hosting where the servers for the individual services hosted on the system are provided by the operator.

3.3 Feedback-based resource schedulers

Existing proportional schedulers (like SFQ [16], Lottery [25], SMART [20], BVT [14], etc.) are capable of proportional allocation of resources among resource principals, but they do not consider application level progress when scheduling resources.

Proportional schedulers assign each resource principal a fraction of the total available resource (e.g., a fraction of the available CPU cycles, or disk bandwidth). These schedulers allocate resources to a demanding principal as long as that principal's resource usage is at or below its assigned share. In addition, a demanding principal is allocated resources that are unused because other principals currently demand less than their share.

Our resource schedulers extend the proportional allocation model by taking application feedback into account. Each principal is assigned a target QoS metric that is reported by the application. In addition, a fraction of the total resources can also optionally be reserved for any principal (as with proportional schedulers). For each resource class r, resources are scheduled between principals based on the following rules given in order of decreasing priority:

1. All principals i such that $(qos_i(t) < target_i \; and \; usage_{ir}(t) \; < \; reservation_{ir})$
2. All principals i such that $(qos_i(t) < target_i)$
3. All remaining principals in round-robin order

The schedulers allocate resources to a demanding principal as long as that principal's reported QoS metric is below target. Therefore, resource allocation is primarily driven by application feedback and allowing a principal to meet its contract is the primary concern. Among these, priority is given to those principals whose resource usage falls below its reservation. This guarantees principals with reservations a minimum system capacity on occasions when the system is incapable of meeting the target of every principal (such a condition would ultimately be detected by the resource monitor). On the other hand, reserved but unneeded resources are made available to meet contracts of other demanding principals. Experiments described in Section 5.1 illustrate the operation and performance of our scheduling framework.

4 Prototype Implementation

In this section, we briefly describe a prototype implementation of our resource management framework for Web server QoS.

A modified version of the FreeBSD-4.0 operating system was used to support the prototype. Resource containers [6] were added to the kernel, and were used to associate separate resource principals with the various services hosted on the system. The prototype supports resource management for the CPU, disk and memory resource classes.

As mentioned in Section 3.1, the main memory resource is partitioned among the services. A fixed number of memory pages can be assigned to a principal (service). Page replacements for a principal only affect the memory pages assigned to it. However, unused memory pages are made available to demanding principals until needed by principals to which they are assigned.

Scheduling of network bandwidth is not supported by our prototype. In our experimental setup, available network resources were never a limiting factor; therefore, network bandwidth allocation was not needed. Adding support for managing network bandwidth on the server's interfaces would be straightforward and similar to the way the CPU and disk bandwidth resources are managed.

Resource containers [6] add about 6k lines of code to FreeBSD-4.0 kernel. This code includes the proportional-share schedulers for CPU and disk as well as support for memory partitioning. Implementing admission control and resource monitoring required 1k lines while implementing the feedback-based CPU and disk schedulers added another 700 lines of code.

The Apache-1.3.12 [3] webserver was used to host the Web services. We modified Apache slightly to report its QoS metric to the kernel periodically using a new system call. The QoS metric used in all experiments was average throughput measured in requests per second. While we have not experimented with average response time as a QoS metric, we are confident that only modest additions to our implementation are required to support it, mainly to enable the trial system to vary resource allocations and measure their effect on the response time.

4.1 Implementation of feedback-based schedulers

Our implementation of the feedback-based CPU scheduler is based on lottery scheduling [25]. This proportional share policy was extended so that application feedback is used to maintain contractual QoS targets. The scheduler allocates the CPU among resource containers, which represent the different services hosted on the system. In implementing the feedback-based disk scheduler, we similarly extended the start-time fair queuing (SFQ) [16] policy. This scheduler was used to schedule disk requests from various resource containers in the system.

Our implementation of feedback-based CPU and disk schedulers supports hosting services in trial mode on a live system with other services already under contract. The service being considered for admission control runs in best-effort mode, i.e., resources are only allocated to it once the contractual targets of all other services have been met.

4.2 Implementation of the Resource Monitor

The resource monitor is implemented as a Unix application process. It periodically uses a system call to sample the average resource consumption of each service for every resource class. A 24-hour period is divided into fixed length intervals (1 minute in our prototype) and samples are maintained for each interval. During each interval, the average resource needs per service for every resource class are recorded. The sampled resource needs for the past 5 days are maintained for each interval. Thus, about 58 KB of memory space is needed for a service to store samples for the CPU and disk resource classes (the implementation does not sample other resource classes). The memory requirements for storing the samples are, therefore, reasonably small. Based upon the stored samples, for each interval the monitor determines the danger of contractual violations using equation 4. As discussed in Section 3.2, the probability is estimated from the relative frequency of the event when the sum total of the resource needs are less than 90%.

While we implemented our prototype on a single-node server for simplicity, the concepts presented in Section 3 are applicable to any system that supports the notion of a resource principal. For example, our work could be applied to a system based on a cluster of workstations where the resource principals are Cluster Reserves [4].

5 Experimental Results

In this section, we present performance results obtained with our prototype implementation. Our results are based on both synthetic as well as real traces derived from Web server logs. In all experiments, throughput as measured in connections per second (same as requests per second for HTTP/1.0) was the QoS metric.

As mentioned in Section 4, a slightly modified Apache-1.3.12 Web server was used, running on a FreeBSD-4.0 kernel, extended with resource containers and with our framework for measurement-based QoS. All experiments were performed on a 500MHz Pentium III machine configured with 1 GB of main memory and a HP SCSI-II disk. The Web requests were generated by a HTTP client program designed for Web server benchmarking [5]. The program can generate HTTP requests from synthetic or real logs either as fast as the Web server can handle them or at a rate dictated by timestamps in the log. Seven 166 MHz Pentium Pro machines were used to run the client program.

The server machine and the seven client machines were networked via a 100Mbps Ethernet switch. Available network bandwidth was not a limiting factor in any of the experiments reported in this section.

5.1 Feedback-based schedulers

The first experiment demonstrates the operation of our CPU scheduler. Three services were hosted on our prototype and CPU reservations of 40%, 20% and 40% were made for them, respectively. The client programs generated requests for the first two services, while service 3 did not have any load. A synthetic workload was used where all requests are for a single file of size 6KB. The rate of request generation matched the capacity of the server.

Figure 1 shows the throughput and CPU usage of both services over time. For the first 20 seconds of the experiment, no contractual targets were set for the services. In this phase, our scheduler behaves like a conventional proportional scheduler and distributes CPU resources between services 1 and 2 in proportion to their reservations, i.e. 2:1.

After 20 seconds, contractual targets of 1000 conn/s and 200 conn/s were set for service 1 and 2, respectively. This results in an allocation of 80% and 20% to the respective services. The reason is that service 2 can meet its contractual target with its reservation of 20%, while service 1 needs additional resources. The scheduler realizes that the 40% resources reserved for service 3 are available, and provides them to service 1. Even at a 80% CPU allocation, service 1 remains unable to meet its contractual target of 1000 conn/s. Since no more resources are available, the throughput of service 1 gets limited to 800 conn/s.

At 40 seconds into the experiment, the contractual targets of services 1 and 2 are reversed – i.e., they are assigned values of 200 conn/s and 1000 conn/s, respectively. The scheduler reacts accordingly and as a result, the CPU

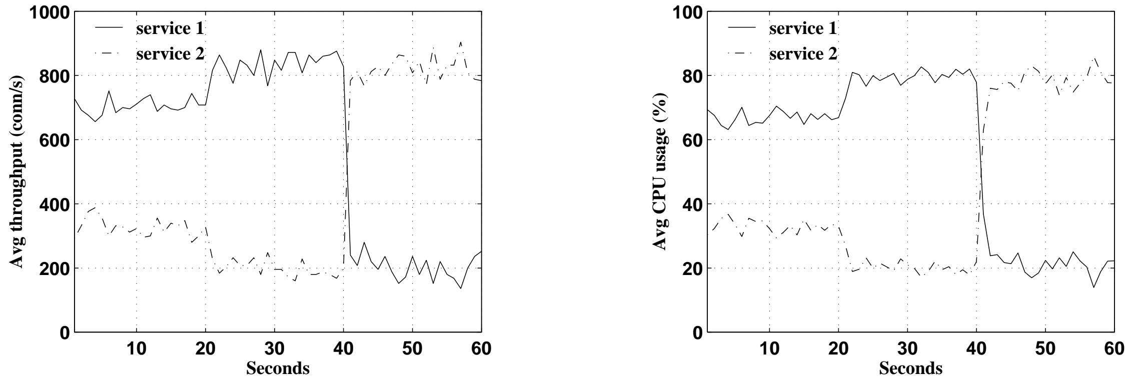

Figure 1: **Feedback-based CPU scheduling**

allocations of services 1 and 2 become 20% and 80%, respectively. This is because service 1 can meet its contractual target of 200 conn/s with 20% of the CPU resources. The remainder of the 40% reserved for service 1 and the 40% reserved for service 3 are considered available. Therefore, service 2 receives these 60% of the CPU resources and its effective allocation becomes 80%.

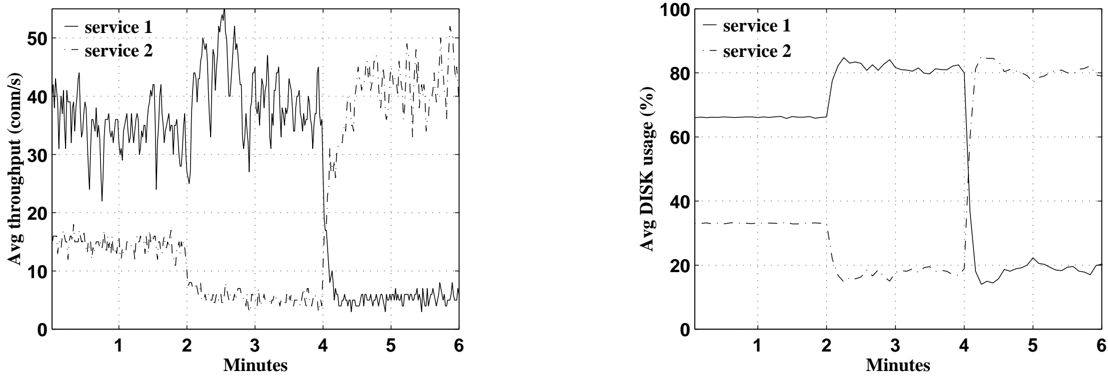

Figure 2: **Feedback-based disk scheduling**

Figure 2 presents results from a similar experiment designed to demonstrate the operation of the disk scheduler. A trace-based workload was used in this experiment (i.e., requests to distinct files selected from the IBM trace described below). Reservations of 40% and 20% were made for two active services that both generate a significant disk load. Targets of 40 conn/s and 5 conn/s were assigned after 2 minutes and reversed after 4 minutes of experimental time. Like the CPU scheduler in the previous experiment, our disk scheduler adjusts the disk allocation in order to meet the respective targets. The relatively high variability in the throughput is due to large variations in the sizes of requested files.

5.2 Admission control

The next set of experiments is designed to evaluate our admission control framework. Trace-based workloads were used in these experiments. Our traces were derived from logs of four Web servers: (i) the Anonymous Computer Science departmental server; (ii) the server for the 1998 Soccer World Cup; (iii) IBM corporation's main server (www.ibm.com); and, (iv) Google's main server (www.google.com). The Anonymous trace spans a period of 15 days in March 2000 and contains requests for 15000 distinct files with a dataset of 1.13 GB and an average request size of 34 KB. The trace from the Soccer World Cup covers 6 days and contains requests for 5163 distinct files with a dataset of 89 MB and an average request size of 6 KB. The IBM trace spans a period of 6 days and contains requests for 38500 distinct files with an average request size of 3 KB. While the above traces consist mainly of requests for static documents, the trace from Google consists solely of CGI requests for dynamic content. This trace spans a period of 6 days with an average response size of 12 Kbytes and an average response time of 0.721 seconds. More information

about these traces can be found in the Appendix.

To generate suitable values of load on the server and also to reduce experimental runtime, we modified the timestamps in the original traces in such a way that the load was scaled in the various traces by different factors, while simultaneously preserving the synchronization between the traces with respect to the daily load variations. Our client program played requests from the processed traces based upon the modified timestamps.

To achieve this compression of the workload in time, one can take several approaches, such as taking future requests in the original trace or repeating requests in the same period to supplement the load in the processed trace. All of these approaches change the characteristics of the original trace in subtle ways. However, we explored both approaches and found that the results of our experiments were virtually unaffected by this choice.

A large number of experiments were performed to test the admission control framework. Due to space limitations, we are only able to present a selection of results.

5.2.1 CPU as bottleneck resource

In the first experiment, services corresponding the the World Cup trace and the IBM trace are hosted on the live system as predictive services with contractual targets of 200 conn/s and 450 conn/s, respectively. The service corresponding to the Anonymous trace is considered for admission in this system and is made to operate in trial mode for this purpose as a prospective predictive service. The resources on the server were configured so as to make CPU the bottleneck resource. That is, the memory was partitioned so as to comfortably fit the working sets of each of the services in memory.

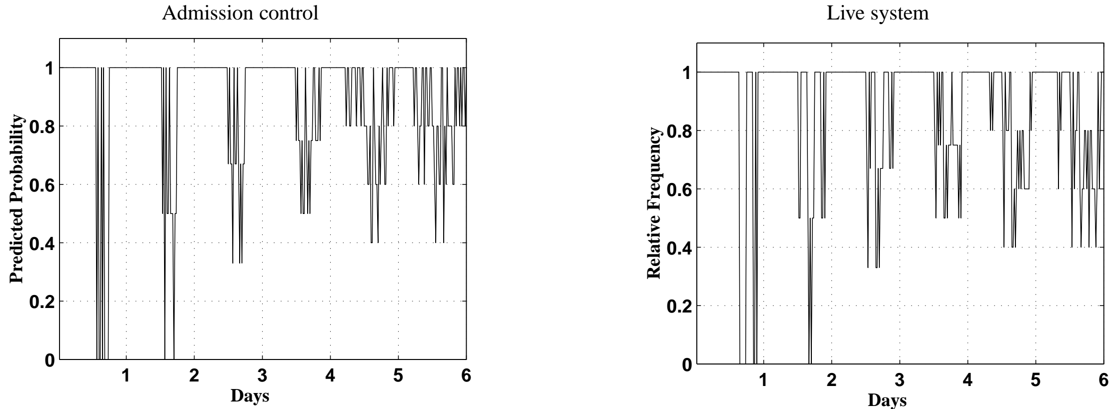

Figure 3: **CPU:** $target_{WC}$ **= 200,** $target_{IBM}$ **= 450,** $target_{Anonymous}$ **= 100**

We first consider a contractual target of 100 conn/s for the Anonymous trace. Based on the resource usage on the idle machine, and on the collective usage of services on the live system, our admission control framework computes the probability from equation 1 at one second time intervals (the value of $thresh$ is 90% and the value of C is chosen to be 0.75). The left plot in Figure 3 shows results from the admission control framework indicating that the service cannot be admitted into the live system with a contractual target of 100 conn/s (as there are times when the probability is less than $C = 0.75$). The right plot in Figure 3 shows results from the resource monitor when that service was admitted despite the rejection of our admission control framework. It plots the relative frequency of the event when the collective resource needs of all services are less than 90%. The close agreement between the plots shows the predictive power of our admission control framework.

We next consider admitting the service corresponding to the Anonymous trace with a reduced contractual target of 24 conn/s. The left plot in Figure 4 shows the probability as computed by our admission control framework and indicates that the service can be admitted into the live system. The right plot in Figure 4 shows results produced by the resource monitor while hosting the service on the system. The two plots closely agree with each other, as before.

Next we consider whether the three services could be hosted jointly on the server as guaranteed services. From the CPU usage and throughput on the idle prototype, we computed the maximum CPU resources required for meeting the contractual targets for each of the three services. These were 66.50%, 42.68% and 55.66%, respectively, for the Anonymous, World Cup and IBM traces. As these total up to more than 100%, this implies that not all three services can be hosted on our prototype as guaranteed services. This demonstrates that the weaker contracts of predictive services allow higher system utilization, while still maintaining contracts with high probability.

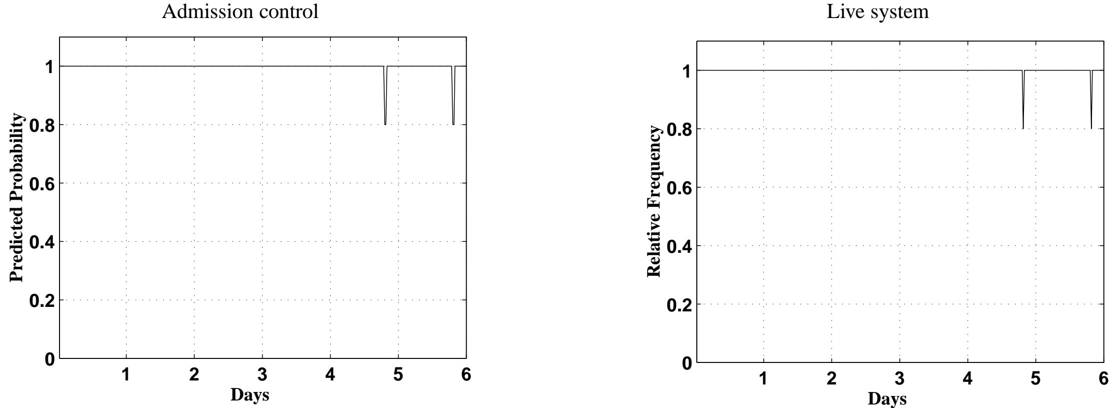

Figure 4: **CPU:** $target_{WC}$ **= 200,** $target_{IBM}$ **= 450,** $target_{Anonymous}$ **= 24**

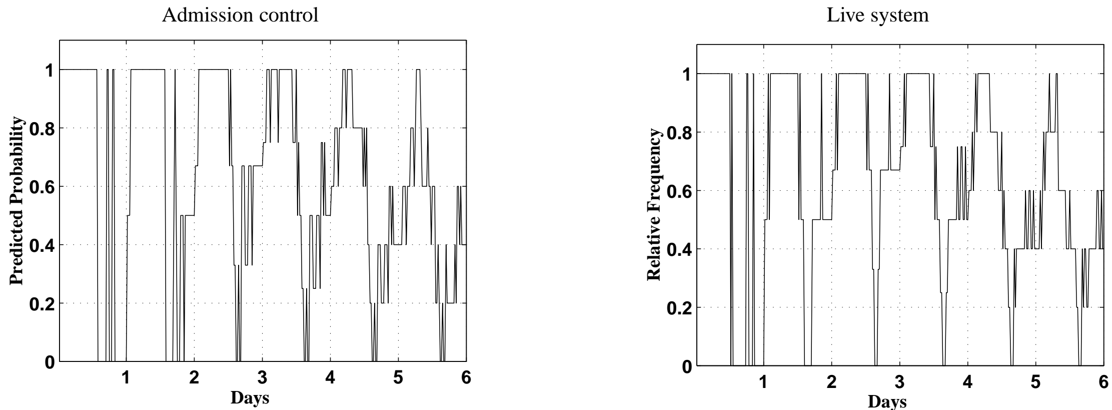

Figure 5: **CPU:** $target_{Anonymous}$ **= 100,** $target_{WC}$ **= 250,** $target_{Google}$ **= 50**

5.2.2 CPU and Disk as bottleneck resources

We next performed an experiment where the server was configured so as to render both the CPU as well as disk resources limiting factors in the achieved throughput. Also, serving dynamic content requests involves the execution of arbitrary server applications and often involves database queries and updates. Therefore, both the amount and the variance of resource consumption per request tends to be much larger for dynamic than for static content. To ensure that our resource management framework is effective under such conditions, we used a workloads that contains CGI requests.

The Google trace consists of dynamic content requests. In our experiment, requests for the Google service were handled by a synthetic CGI program. For each request, the CGI program responds with content of a size corresponding to the size specified in the trace. Moreover, for each request, the program consumes CPU time corresponding to the response time specified in the Google trace, and for requests with response times larger than 10 milliseconds, the program additionally performs an 8 KB disk access.

The services corresponding to the Anonymous trace and the World Cup trace were hosted on the live system as predictive services with contractual targets of 100 conn/s and 250 conn/. The service corresponding to the Google trace is considered for admission into this system. In order to stress the disk subsystem, the memory was partitioned so as to allocate an amount of memory that is only slightly larger than the working set of each of the static services. Thus, 30 MB and 20 MB were assigned to the Anonymous and World Cup services, respectively. 60 MB of memory were assigned to the service corresponding to the Google trace.

A contractual target of 50 conn/s was first considered for the Google trace. The probability plots produced by our admission control framework are shown in Figures 5 and 6 for the CPU and the disk resource class, respectively. As the probability falls below $C = 0.75$ several times in each case, the service cannot be admitted into the live system with a target of 50 conn/s. Figures 5 and 6 also contain results from the resource monitor when the service is admitted despite rejection by the admission control framework. These depict the relative frequency of the event when the collective resource needs are less than 90%. Again, the close agreement between the plots from the admission

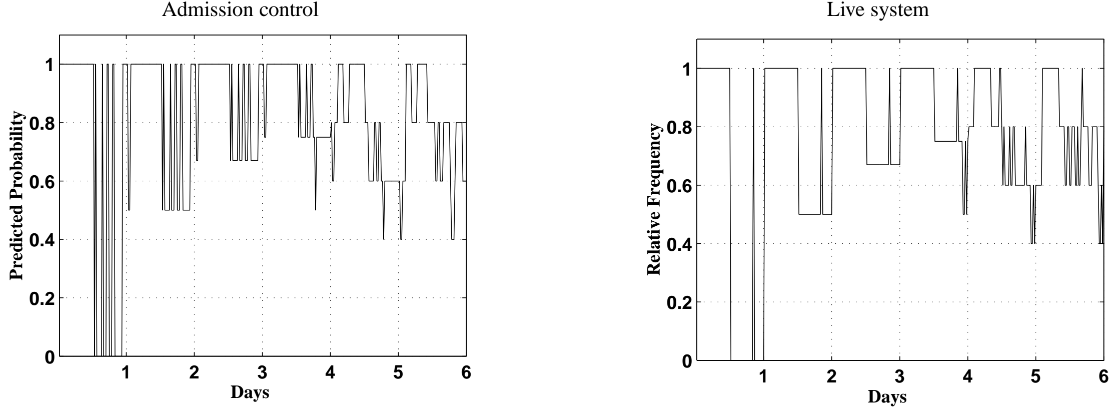

Figure 6: **Disk:** $target_{Anonymous}$ **= 100,** $target_{WC}$ **= 250,** $target_{Google}$ **= 50**

control and the resource monitor shows that the admission control framework is capable of accurately characterizing the live system under this diverse workload.

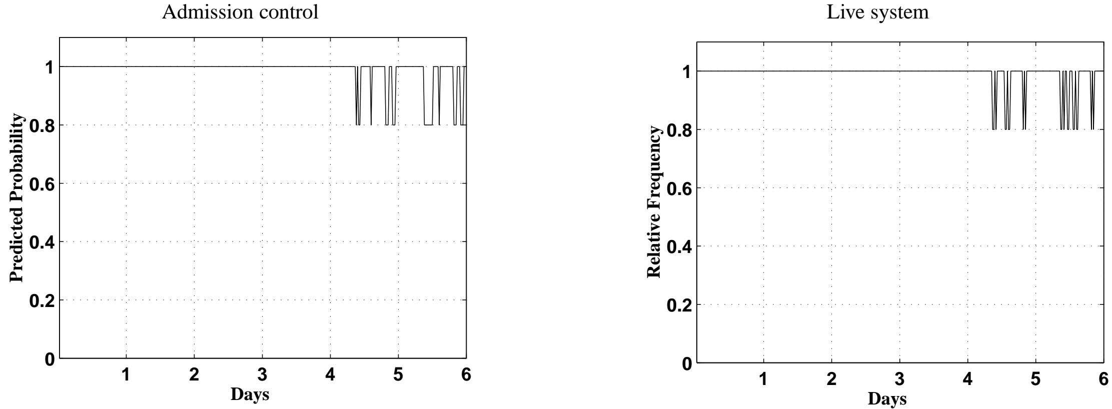

Figure 7: **CPU:** $target_{Anonymous}$ **= 100,** $target_{WC}$ **= 250,** $target_{Google}$ **= 5**

The service corresponding to the Google trace was then reconsidered for admission with a target of 5 conn/s. Figures 7 and 8 show the results for the CPU and the disk resource, respectively. The plots from the admission control framework indicate that the service can be admitted into the live system. (The value of the probability remains above $C = 0.75$ at all times). Again, the results produced by the resource monitor after hosting the services on the live system closely agree with those from the admission control framework.

5.3 Resource Monitor

We next present an experiment to demonstrate the operation of our resource monitor (Section 3.2). With the help of artificially varying load patterns introduced in synthetic traces, we show the detection of situations that lead to contract violations.

We hosted two services on the server, one predictive and the second guaranteed. The traces containing the requests for the services were produced by synthetically generating timestamps to resemble the load characteristics typical of real Web servers.

The trace for service 1 was such that after two days, the peak load increases unexpectedly and significantly (enough to consume all available CPU resources). The contractual target for the first service (predictive) was set to 1000 conn/s. The contractual target for the second service (guaranteed) was 500 conn/s and it was given a CPU reservation of 55%, which was sufficient to meet its peak demand.

The first two plots in Figure 9 show the variation of throughput and CPU usage with time. The last plot in Figure 9 depicts the relative frequency of the event where the measured collective resource needs of all services are less than $thresh$ (as computed by equation equation 4). This value is computed from samples of resource needs taken every second in the experiment. The values for $thresh$ and C were chosen to be 90% and 0.75, respectively.

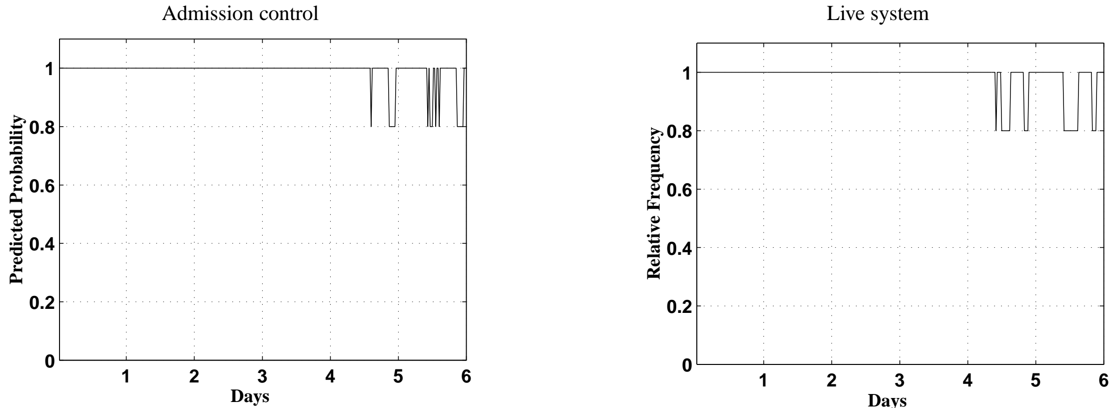

Figure 8: **Disk:** $target_{Anonymous}$ **= 100,** $target_{WC}$ **= 250,** $target_{Google}$ **= 5**

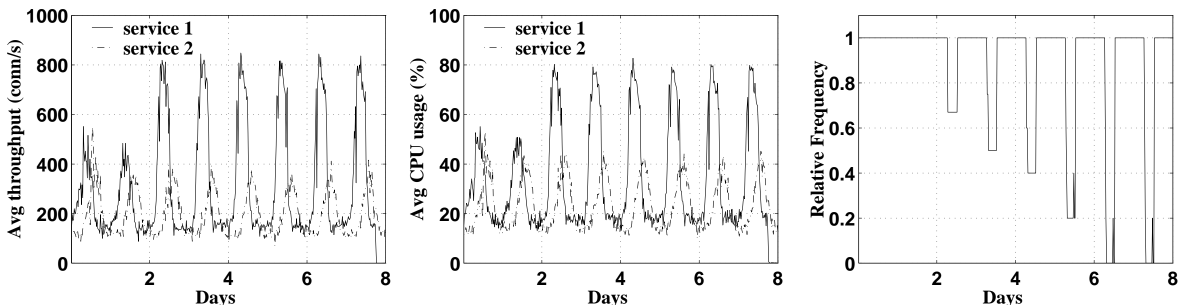

Figure 9: **CPU:** $target_1$ **= 1000,** $target_2$ **= 500**

The results from Figure 9 indicate that for the first two days of the experiment, the relative frequency of samples where the collective resource needs are less than 90% of the total CPU resources remains high. However, once the load on service 1 increases, the relative frequency drops down in the corresponding time intervals, indicating that the contract for service 1 can no longer be maintained. Figure 9 also shows that despite the increased load on service 1, the performance of the guaranteed service remains unaffected. This confirms that the contracts of guaranteed services cannot be affected by load variations of other services. On the other hand, contract violations of predictive services can occur, but our resource monitor is capable of reporting this early, so that corrective action can be taken.

6 Related Work

Banga et al. [6] proposed the resource container abstraction that separates the notion of a resource principal from threads or processes and provides support for fine-grained resource management in the operating system. Coupled with LRP [13], resource containers are capable of affording performance isolation on a single node. Other related abstractions are Activities [18] in the Rialto real-time operating system, software performance units (SPU) [24] proposed in the context of shared-memory multiprocessors, reservation domains in the Eclipse operating system [11, 10] and paths in Scout [22]. Cluster Reserves [4] extend resource containers to enable global resource management in a cluster of workstations.

Various scheduling policies for proportional share resource allocation can be found in the literature, including Lottery scheduling [25] and STFQ [16]. The SMART [20] multimedia scheduler integrates priorities and weighted fair queuing to meet real-time constraints while simultaneously supporting non real-time applications. Our feedback-based resource schedulers extends proportional schedulers to achieve performance targets using application feedback.

Jamin et al. [17] describe a measurement-based admission control algorithm for predictive services in packet networks. Our work applies similar concepts to Web server systems. We use measurements to translate application level performance targets into resource allocations in the system and perform admission control for both guaranteed as well as predictive services.

Li and Jamin [19] use a measurement-based approach to provide proportional bandwidth allocation to Web clients by scheduling requests within a Web server. Their approach is not able to guarantee a given request rate or response

time, it may be suitable only for static content and has not been evaluated on trace-based workloads. Moreover, the system cannot be used for capacity planning, i.e., to predict which services can be co-located on a given system.

Bhatti and Friedrich [8] describe modifications to the Apache webserver in order to support differentiated QoS between service classes. Request processing is performed based on a classification of HTTP requests into priorities at the server application. The method is strictly priority-based and proportional allocation of server resources between service classes is not supported.

OS mechanisms such as the resource containers [6] address the problem of proportional resource allocation more generally by fully accounting for all kernel processing. Our work extends and complements this prior work. It seeks to provide both proportional as well as predictable quality of service in Web servers, and provides QoS in terms of application-level metrics such as request rate and average response time.

7 Conclusions

We have presented a measurement-based resource management framework for providing predictable and differentiated quality of service in Web servers. The framework allows Web servers and proxy operators to co-locate virtual sites and Web services, while providing predictable quality of service, in terms of average request rate or average response time.

The framework consists of a measurement-based admission control process that allows operators to determine whether a set of sites can be co-located on the same server system, based on the measured statistics of the sites' resource consumption under its live workload, and its desired quality of service and service class (guaranteed, predictive, or best effort). Once a set of services has been admitted, feedback-based resource schedulers ensure that all sites achieve their QoS targets, while being allowed to use excess resources not currently claimed by other sites.

An empirical evaluation of a prototype implementation shows that the system is able to predict with high confidence if sites can be co-located on a system; that it is able to maintain the target QoS levels of admitted sites with high probability; and, that it is able to achieve high average hardware utilization on the server system.

We expect that the framework will stimulate further research in algorithms for admission control, capacity planning, scheduling and generally, approaches to capacity planning and QoS in Web servers.

References

[1] Akamai. http://www.akamai.com/.

[2] D. Andresen et al. SWEB: Towards a Scalable WWW Server on MultiComputers. In *Proccedings of the 10th International Parallel Processing Symposium*, Apr. 1996.

[3] Apache. http://www.apache.org/.

[4] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, June 2000.

[5] G. Banga and P. Druschel. Measuring the capacity of a Web server under realistic loads. *World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation)*, 1999.

[6] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[7] P. Barford and M. Crovella. Critical Path Analysis of TCP Transactions. In *Proceedings of the ACM SIGCOMM 2000 Symposium*, Stockholm, Sweden, Aug. 2000.

[8] N. Bhatti and R. Friedrich. Web Server Support for Tiered Services. *IEEE Network*, 13(5):64–71, Sept. 1999.

[9] J. Bruno, J. Brustoloni, E. Gabber, M. McShea, B. Özden, and A. Silberschatz. Disk Scheduling with Quality of Service Guarantees. In *In Proceedings of the IEEE ICMCS Conference*, Florence, Italy, June 1999.

[10] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Retrofitting Quality of Service into a Time-Sharing Operating System. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[11] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proceedings of the USENIX 1998 Annual Technical Conference*, Berkeley, CA, June 1998.

[12] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Technical Conference*, Jan. 1996.

[13] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[14] K. J. Duda and D. R. Cheriton. Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, Dec. 1999.

[15] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[16] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating System. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[17] S. Jamin, P. B. Danzig, S. J. Shenker, and L. Zhang. A Measurement-based Admission Control Algorithm for Integrated Services Packet Networks. In *Proceedings of the ACM SIGCOMM '95 Symposium*, Boston, MA, Aug. 1995.

[18] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera. Modular real-time resource management in the Rialto operating system. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[19] K. Li and S. Jamin. A Measurement-Based Admission Controlled Web Server. In *Proceedings of the IEEE Infocom Conference*, Tel-Aviv, Israel, Mar. 2000.

[20] J. Nieh and M. S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, New York, Oct. 1997.

[21] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[22] O. Spatscheck and L. L. Peterson. Defending Against Denial of Service Attacks in Scout. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[23] I. Stoica, S. Shenker, and H. Zhang. Core-Stateless Fair Queuing: A Scalable Architecture to Approximate Fair Bandwidth Allocations in High Speed Networks. In *Proceedings of the SIGCOMM '98 Conference*, Vancouver, Canada, Aug. 1998.

[24] B. Verghese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[25] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994.

A Trace Characteristics

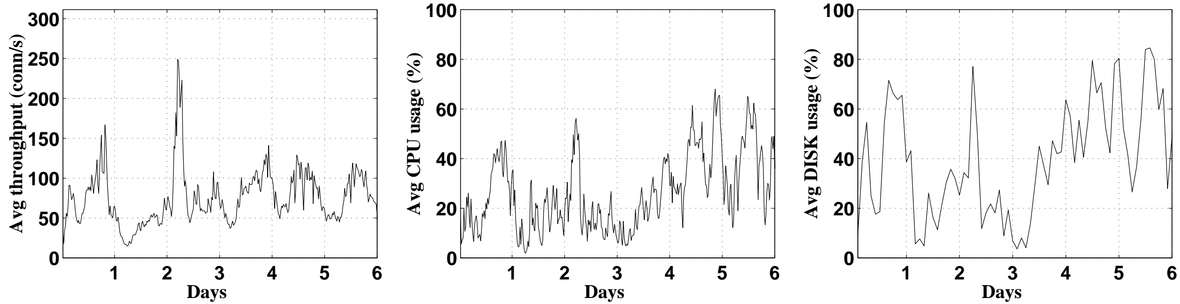

Figure 10: **Anonymous trace**

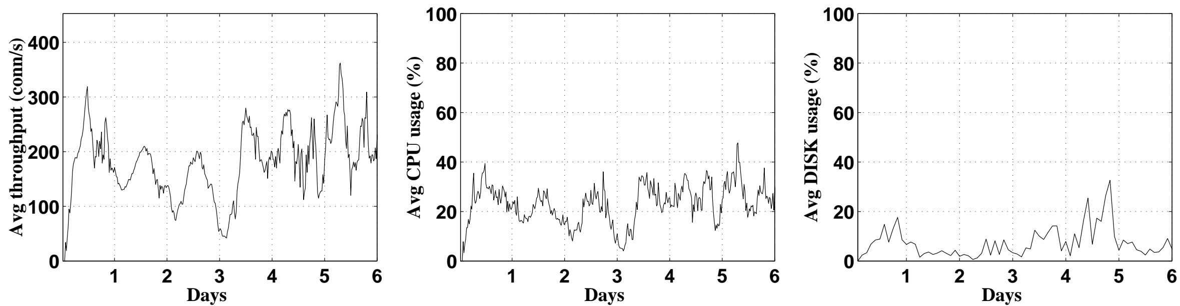

Figure 11: **1998 Soccer World Cup trace**

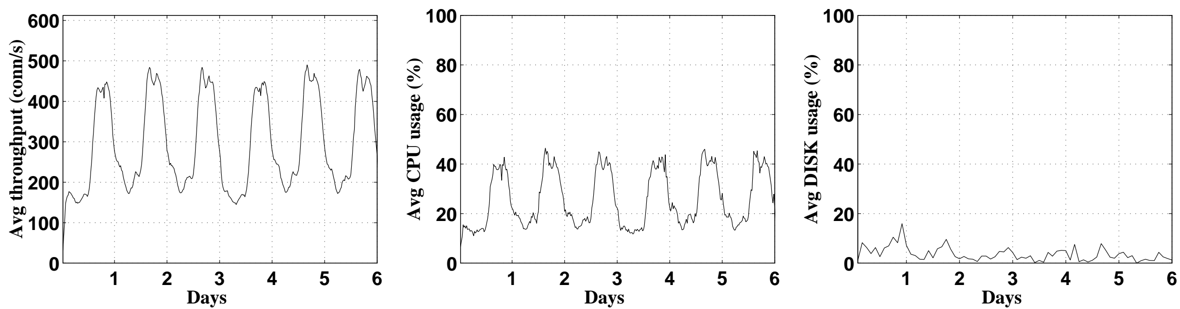

Figure 12: **IBM trace**

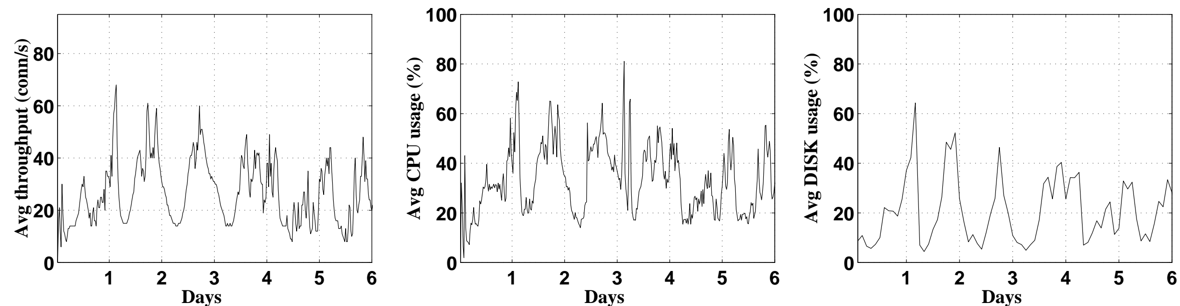

Figure 13: **Google trace**