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

Mohit Aron* Sitaram Iyer Peter Druschel

{aron, ssiyer, druschel}@cs.rice.edu

Department of Computer Science
Rice University

Abstract

This paper presents a resource management framework for providing predictable quality of service (QoS) in Web servers. The framework allows Web server and proxy operators to ensure a probabilistic minimal QoS level, expressed as an average request rate, 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 [2, 17], Web content caching [12, 26], provision of differentiated services in the Internet [20, 31], and differentiated services in Web servers and proxies [4, 6, 7, 10, 22].

This paper focuses on the latter of these problems, namely providing predictable and differentiated quality of service for Web servers and proxies. Of specific interest is the problem of hosting Web sites and Web Services on third-party cluster farms, run by agencies we shall call Web site operators. These operators are often charged with the task of hosting a number of lightweight services alongside some relatively popular ones. They naturally find it cost-effective to co-host several lightweight services on each server node in the cluster, rather than necessarily dedicating one or more nodes per service [32]. Furthermore, shared hosting lowers ISP costs for small-scale content providers, and facilitates the proliferation of diverse but low-traffic sites and services.

Shared hosting is now common practice for static content, and is becoming increasingly popular for dynamic content based services. Web site operators such as One Squared [25], Site5 Internet Solutions [28] and Verizon Online [34] support shared hosting of static and dynamic content. They offer *service contracts* to content providers, typically based on limits imposed on network bandwidth usage and disk quotas, along with different restrictions on the software allowed for the service, the degree of customer support provided, etc. Such contracts convey an approximate sense of different service levels, but their inherent drawback is that they force the content provider into categorizing the service into one of the available categories, depending on its expected activity. If the service subsequently encounters

*Now at Google Inc., Mountain View, CA.

a load burst exceeding the resources allotted in this category, then either the provider is billed much higher, or the load spike is denied service. Content providers can either conservatively pay more and fit into a higher service class, or pay less but suffer poorer service than they deserve. We claim that service contracts in current web hosting systems face the problem of pricing without a sense of the *performance requirements* of the service.

This paper presents a scheme for providing service contracts based on *high-level performance metrics*, such as a guarantee on throughput or response time experienced by requests to the web site or service. The Web site operator may wish to provide a deterministic or probabilistic guarantee of such a level of service – regardless of the load imposed by other requests on the shared Web server machine. When a number of low-traffic services are co-hosted on a machine, each of them can be ensured some minimal quality of service, based on the patterns of their resource usage. The operator can derive pricing strategies for these lightweight services, so that they are more cost-effective for the operator (due to optimized co-location of services) as well as cheaper for the service provider.

To achieve this, we describe a resource management framework that ensures pre-specified throughput targets for Web sites and services. It tolerates varying resource demands (CPU, memory, disk and network bandwidth) of the service, as well as competing activity on the system, as long as the collective resource requirements do not exceed the system's resources. In particular, the framework provides *admission control* for Web server operators to decide whether a given service class can be co-located with other sites currently hosted on the same machine (capacity planning). This admission control phase proceeds as follows: a new service with unknown load and resource demand characteristics is first operated in "trial" (best effort) mode, 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 reasonable period of time.

Our framework extrapolates from this statistical envelope to predict the resource demands needed to guarantee a certain QoS level in the future. The content provider can use this resource usage information to estimate the cost of hosting the service, and thus establish a contract (service level agreement – SLA) with the customer. Our system allows SLAs to be described in terms of an average target throughput, which may be specified to vary based on time of day, day of week, etc. Now the service provider can use the same statistical envelope to decide which services can be co-located on a given server node.

After a set of services are co-located, the resource scheduling framework ensures 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.

This paper thus describes the design of the measurement-based admission control and scheduling framework, and presents results using both synthetic and trace-based workloads (encompassing both static and dynamic content workloads). The results suggest that for various web-based workloads, our framework is effective in predicting 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 using the prototype. Related work is described and compared against in Section 6, and Section 7 concludes.

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 response times are increasingly dominated by server delays.

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 (a) the contracted QoS levels are maintained with high probability, and (b) the average hardware utilization is high.

Web hosting operators typically employ cluster-based Web farms to host various third-party services. Services are hosted in either a dedicated or a virtual manner; the latter approach dynamically shares cluster nodes among multiple services, and enables flexible resource sharing and higher hardware utilization. Products such as ServerXchange from Ensim [16] provide the basic mechanisms for web hosting in both the above forms. Such solutions generally provide best-effort service; there is no support for predictable quality of service using metrics such as average throughput or average response time bound.

Contributions: There have been several recent research papers covering different aspects of the predictable web server QoS problem [32, 27]. Our paper, in contrast, addresses this problem from a more *systemic perspective*, and solves several key problems required to make the concept practical. This includes support for multiple resource classes (such as CPU, disk and memory) – taking into account the interactions between them, a comprehensive admission control and resource monitoring scheme, and an evaluation with co-hosted CGI-based workload with bottlenecks along multiple resources. Specific comparisons to related work are made in Section 6 (especially in Section 6.2), where each of these contributions is discussed in detail.

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* to decide whether a set of services with associated contracts can be accommodated in the system, (ii) a *resource scheduling framework* to ensure that contracts are met, despite varying resource demands (assuming hardware resources are not exceeded), and (iii) a *resource monitor* to foresee impending contract violations on the live system, and warn the operator in advance.

We begin by defining some terminology. "Provision of predictable quality of service" is 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.

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.

An *operator* manages systems (servers and proxies) that are used for hosting services on behalf of one or more *content providers*. The content provider supplies 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. Without loss of generality, we assume that larger values of the QoS metric are more desirable. Example QoS metrics are (1) the average number of successfully completed requests per second, or (2) the reciprocal of the 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, given sufficient demand for

the service, the system is unable to provide sufficient resources to the service in order for it to meet its contractual target.

Our resource scheduling framework trusts the services hosted on the system to report the truthful 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, rather than the content provider. However, note that the QoS metrics reported by different services *need not be comparable*; it is enough for each service to have its QoS metric and its target on the same scale.

The *service type* determines the strength of a contract. For *guaranteed services*, a system must make every effort to 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.

CPU time, main memory pages, disk bandwidth and network bandwidth each constitute a *resource class*. For simplicity, we first describe the system assuming that only one resource class (such as CPU) is possibly in contention, and that other resources are always plentifully available. Section 3.4 extends this system to consider multiple resource classes, and examines their individual idiosyncrasies.

The rest of this section describes the admission control framework, the resource monitor, and the feedback-based scheduling framework, which constitute 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 fixed 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 needs to have plenty of uncommitted resources. This may be either a separate idle system set aside for this purpose, or may be a normal *live* system that continues to host other services under contract. In the latter case, the service under test is a best-effort service; thus the resource scheduling framework ensures that it does not interfere with the live services. The trial system is assumed to have the same hardware characteristics as the system in which the service is to be admitted.

The trial service is then subject to its regular live workload. The resource usage (of the only resource class currently in consideration) and the QoS metric (e.g. throughput) are monitored for the service, over a period of time that is deemed long enough for determining 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.

Resource needs:

The load on many Web sites tends to follow a periodic pattern, with a fixed period T (e.g., 24 hours or 7 days). 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 on the characteristics of the client load. A daily or weekly period

usually proves sufficient to capture significant patterns of variability in many web workloads, as can be seen from the traces in the Appendix.

We then proceed to compute the *resource need* for the service to maintain its contracted target. Instead of a constant or an arbitrarily varying resource need, we take advantage of the approximate periodicity of the workload to compute the resource need over every time interval in the period. We thus characterize the average resource need during the interval t of service i by a family of random variables $U_i(t)$ (in units of resource usage). We sample these variables as follows.

For service i, let $qos_i(t)$ denote the QoS metric reported by the service during interval t, and let $target_i$ denote its contractual target. Moreover, in each time interval t, the resource scheduling framework reports the resources used by service i as $usage_i(t)$, expressed as a percentage of the total resource availability.

We make a *linearity assumption* on the nature of the service and the resource class, in that the QoS metric is approximately linearly related to the resource usage of the service. If a single resource (such as CPU) were bottlenecked, then given sufficient load, we assume that increasing the amount of the available resource will proportionally increase its throughput. Note that such linearity holds only for resources such as CPU and not for resources such as main memory; this is examined in Section 3.4. Strict linearity is not necessary, since the feedback-based scheduler detects and compensates for deviations.

When the QoS metric is above the target, this assumption allows us to scale down the resource usage to speculate on the amount of resources sufficient to meet the target. When the QoS metric is below the target, then recall that this service is running in trial mode and is provided sufficient resources; therefore, its current resource consumption is enough for the load during this interval. Formally, the resource need, i.e., the value of the random variable $U_i(t)$ is sampled as follows:

$$U_i(t) = \begin{cases} usage_i(t) & qos_i(t) \leq target_i \\ usage_i(t) * target_i / qos_i(t) & otherwise \end{cases}$$

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 resource need does not include these excess resources consumed when the QoS metric exceeds the contractual target.

Admission control criteria:

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:

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

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.

$$\max_t \hat{U}_S(t) + R \leq 100 \tag{2}$$

where $\hat{U}_i(t)$ is the maximal observed resource need of service i during interval t, and

$$R = \sum_{i \in G} \max_t \hat{U}_i(t) \tag{3}$$

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 $\max_t \hat{U}_S(t)$ are reserved, 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 for a modest number (up to ten) of co-located services. Further work may be needed to develop adaptive methods for setting thresholds appropriate for larger numbers of co-located services.

The probability in equation 1 is estimated by approximating the probability distribution of the random variable $U(t) = \sum_i U_i(t)$ by the distribution of the observed samples of $U(t)$. For example, the probability $P(U(t) \leq 90\%)$ 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. Observe that 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 impending danger of violating contracts. Corrective actions may consist of adding more resources to the system (e.g., by upgrading the hardware), or moving some services to other server (or proxy) nodes, or in extreme cases of overload, even 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:

$$P(\sum_{i \in P \cup G} U_i(t) \leq thresh) \geq C \tag{4}$$

The probability in equation 4 is estimated from the relative frequency [1] 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 in danger of violating contracts. The operator of the system can choose to take corrective action after one or more violations of equation 4.

3.3 Feedback-based resource schedulers

Once a set of services is admitted and co-located onto the server, then in high probability, it is *feasible* to allocate resources so as to meet all the QoS targets. It is now the task of the resource schedulers to achieve this goal.

We adopt the *reservation-based resource scheduling* paradigm as the underlying scheduler substrate, since it provides the flexibility needed to overlay our high-level QoS targets. Reservation-based schedulers such as STFQ [18], Lottery [36], SMART [24], and BVT [15] guarantee a specified fraction of the system's resources to each resource principal. Thus, they allocate resources to demanding principals as long as their resource usage is below their assigned reservation. In addition, they allocate unused resources among all demanding principals.

We extend these schedulers to consider application-level QoS metrics, by preferentially allocating more resources to services that are falling below their QoS target. Conceptually, resources are scheduled between principals (after each scheduler quantum), based on the following rules presented in order of decreasing priority:

1. All principals i such that $(qos_i(t) < target_i \; and \; usage_i(t) < reservation_i)$

[1] 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.

2. All principals i such that $(qos_i(t) < target_i)$
3. All remaining principals in round-robin order

We now present a simple interpretation of this feedback-based scheduling policy as applied to the co-hosted services. Our implementation of this policy uses an unmodified reservation based scheduler, but periodically recomputes the *effective reservation* for each service based on its QoS metric and target. Thus, it estimates each service's *resource need* required to meet its QoS target, and assigns it a reservation proportional to this need. Granting the expected feasibility of resource allocation, this simple policy enables all targets to be met. Thus, resource allocation is primarily driven by application feedback. Note that the scheduler uses the currently measured resource utilization, QoS metric and the target, and specifically does not make use of the speculative resource envelopes that were previously recorded for admission control purposes.

The scheduler also allows the operator to optionally specify a *base reservation* value (such as 10%) for each service. This is a fail-safe reservation level that is guaranteed to each service that does not meet its respective target, regardless of whether other services' targets are met. This feature is useful if some service experiences an unexpected burst of load, whereupon it tries to monopolize the system's resources. The resource monitor would presently detect this overload and flag the services as having overshot the system capacity. In the interim prior to any restorative action, the operator-assigned base reservation settings would guarantee a minimal level of progress to other services.

Experiments described in Section 5.1 illustrate the effectiveness and performance of this scheduling framework.

3.4 Supporting multiple resource classes

So far, our system has managed a single, abstractly identified resource class, thus implicitly assuming that the other resource classes are never in contention.

We now extend the system to support multiple resource classes. This extension is important in practice, because resource classes often simultaneously (in combination) and significantly affect system performance, as exemplified in Section 5.2.2. It therefore impacts admission control decisions and adherence to QoS contracts.

This extension to multiple resources may simply seem like applying the equations for computing resource need, performing admission control, and scheduling resources independently to each resource class in consideration. However, there are three interesting obstacles that hinder this simple approach, rendering the extension non-trivial and challenging. Some of these issues arise from some fundamentally unsolved resource management problems; we address them here with practical-minded solutions, and demonstrate their effectiveness in Section 5.

This extension is novel, since none of the following three issues have been addressed or solved by previous work in this context.

1. Memory management through partitioning

Resource classes can be categorized into *time-scheduled* (e.g., CPU time, disk bandwidth, network bandwidth) and *space-scheduled* (e.g., main memory, disk space, file descriptors). The latter category (with focus on the main memory resource class) does not satisfy the linearity assumption postulated in Section 3.1. This is because applications often need a main memory allocation corresponding to their working sets in order to run efficiently. Allocating more memory than the working set may increase performance, but does not even approximately linearly scale it up; whereas allocating less memory than needed imposes a drastic performance penalty if the application starts thrashing.

Moreover, main memory management poses another unique problem – it interoperates with, and strongly influences the load on the *disk subsystem* due to paging I/O. The disk resource envelopes measured during the trial phase may therefore be inapplicable to the live system if the memory allocated to the service differs between the two modes.

Thus, a QoS-sensitive memory allocation and paging strategy is required. It is easy to see that the best-effort paging policies implemented in most general-purpose operating systems (such as Working

sets [13] or Clock) are incapable of providing the fine control needed for predictable QoS guarantees. Furthermore, straightforward memory partitioning mechanisms such as in [19] and [33] are insufficient owing to their inability to judge the appropriate memory partition size that meets our high-level performance goals. Finally, our feedback schedulers based on resource reservations cannot be directly applied to main memory, due to its non-linearity and its interaction with disk I/O.

Our solution is based on partitioning main memory according to the working sets of the services. During the trial phase for a service, our system runs it for a short duration, and determines the tentative amount of main memory required to capture 95% of its requests. This approximately corresponds to the point where the service exhibits a knee in the disk utilization.

For the rest of the trial period, it reserves this amount of memory for the service, limits its physical memory availability to this amount (so that disk utilizations are predictable), and continues the trial period measurements. These services are hosted on the live system if their memory requirements do not exceed the system's main memory – thus achieving admission control for main memory. Free system memory may – or may not – be allocated on a demand basis to the services; doing so may cause conservative admission control decisions, since the memory allocation in excess of the working set may slightly reduce the actual paging I/O below the predicted value. For clarity of experiments, and to avoid the problem of having to quickly preempt this surplus allocation (by potentially paging out memory), our implementation refrains from allocating the leftover memory until the working set shifts.

Over time, both during the trial period and during live operation (in the resource monitor), the working set of each service is periodically recomputed to accommodate for shifting working sets (especially if QoS targets are repeatedly missed). Significant deviations may induce our system to alter the memory partitions, check its admission control criterion by verifying if the working sets fit in memory, and if so, re-measure the ensuing disk utilizations.

This strategy effectively eliminates the interaction between disk and memory resource classes, by fixing the memory partitions. It also ensures that each service receives the amount of memory it needs for reasonable performance.

2. Independently scheduled resource classes

Even among time-scheduled resource classes such as CPU, disk and network bandwidth, there is a problem in using the feedback scheduler independently for each resource, without heed for the allocations of other resources.

The feedback scheduler operates by reserving the amount of resources expected to be necessary for meeting the QoS target. If the QoS metric falls below the target, then it would indiscriminately allocate more of *each resource class* to that service, causing excess allocation of other resource classes. At first this may seem innocuous – since a surplus allocation of a non-bottlenecked resource to a service would only result in the excess being unused by the service.

However, closer scrutiny reveals situations where independently scheduling resources *does* have an adverse impact on QoS contracts. Consider three services A, B and C, where A and B are CPU-bound, and C is completely disk bound (and naturally has a much lower QoS target than A or B). In addition, B has some disk-intensive requests that are serviced so much more slowly than its CPU-bound requests that they hardly contribute to its throughput. Thus, we have constructed a realistic scenario where both CPU and disk may both become saturated. Now if B falls below its target throughput, then the feedback scheduler will kick in, and allocate B more CPU as well as disk. This would steal disk resources from C, and lower its throughput – and potentially even violate its contract. If that happens, the resource monitor would fail admission for the three services.

A more intelligent scheduler could have, in theory, identified the bottleneck that matters to B (viz., CPU), and merely have provided more CPU to B, thus managing to accommodate all three services. Scheduling with bottleneck recognition is a hard problem (with some possible solution approaches not discussed here); note that it can satisfy all three contracts in the above example.

However, observe that satisfying contracts for all three services in this manner implies that a significant and sustained fraction of the disk-intensive requests of service B *do not* receive enough disk bandwidth to get quickly serviced. These requests suffer large response times, and may even be dropped. This *technically* satisfies the QoS metric based on service throughput, but would probably be considered

unacceptable from the content provider's point of view. Hence – in the spirit of the contract – we argue that the above combination of services should not be admitted into the system. It is possible to formally incorporate such a response time consideration into the contract by, for example, requiring that a *utility function* of throughput and average response time be used as the high-level performance metric [27].

Interestingly, note that this desired behaviour (of admitting services only if their throughput targets are met *and* they a reasonable per-request response time) is precisely the behaviour achieved if the time-scheduled resource classes were scheduled independently of each other – just as we originally considered. Then the admission control would be occasionally more conservative than the strict contract specification, in the manner desired. Thus, we conclude that the conceptually clean model of independently scheduling resource classes is, in this context, semantically preferable to bottleneck-identifying schedulers.

3. Disk interference injection

Estimating the disk resource needs during the trial phase 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; this interference was experimentally verified to be significant.

To account for this, our disk scheduler incorporates a novel feature that does not permit services in trial mode 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, then a disk request for a random disk block is injected into the disk queue and scheduled. 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 approximately reflect those observed when hosted on the live system – with sufficient accuracy (as demonstrated in Section 5.2.2) to make meaningful admission control decisions.

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 3.4, 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 currently not supported by our prototype. In our experimental setup, available network resources were never a limiting factor; therefore, 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 yet experimented with average response time as a QoS metric, we expect 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 [36]. 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 (STFQ) [18] 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%.

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 server CPU saturated at 1000 conn/s for this workload). The rate of request generation matched the capacity of the server.

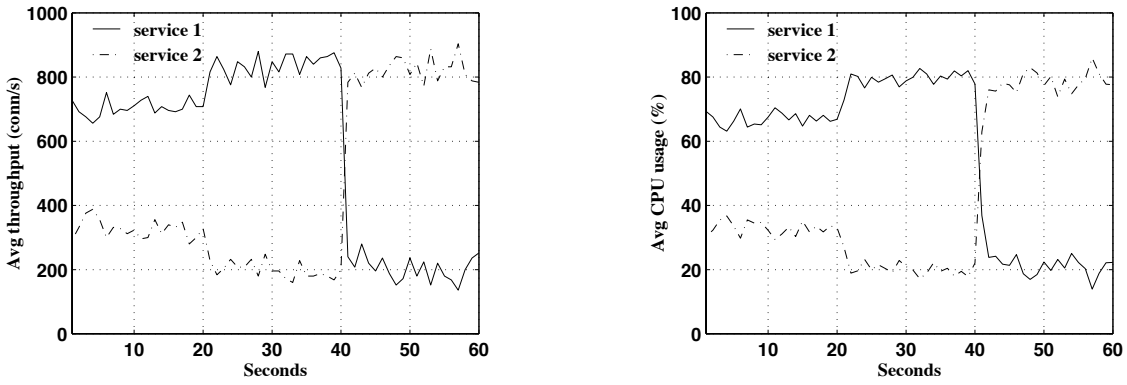

Figure 1: **Feedback-based CPU scheduling**

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 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%.

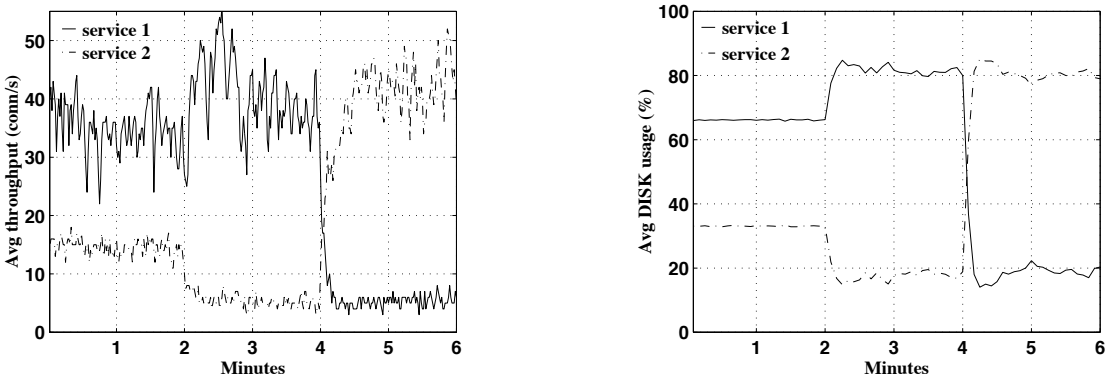

Figure 2: **Feedback-based disk scheduling**

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 the access logs of four Web servers: (i) the Rice University 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 Rice 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. The compression used in our experiments was such that a day's worth of trace data was mapped to a minute of experimental time (the load patterns were preserved in this mapping).

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. No CPU resources are reserved for these services (resource reservations are only made for guaranteed services). The service corresponding to the Rice 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.

We first consider a contractual target of 100 conn/s for the Rice 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 [2] (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%. This agreement between the plots shows how the admission control framework can reliably predict behaviour on the real system. We also have evidence to demonstrate its predictive power for future requests from the same trace; however, we have not explored this aspect in sufficient detail yet.

[2] As one minute of experimental time corresponds to one day of trace data, a one second interval of experimental time corre-

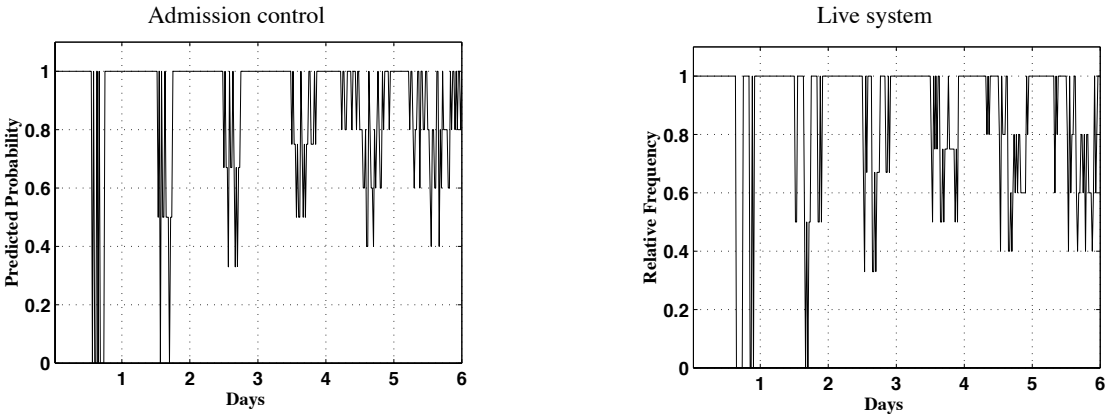

Figure 3: **CPU:** $target_{WC}$ **= 200,** $target_{IBM}$ **= 450,** $target_{Rice}$ **= 100**

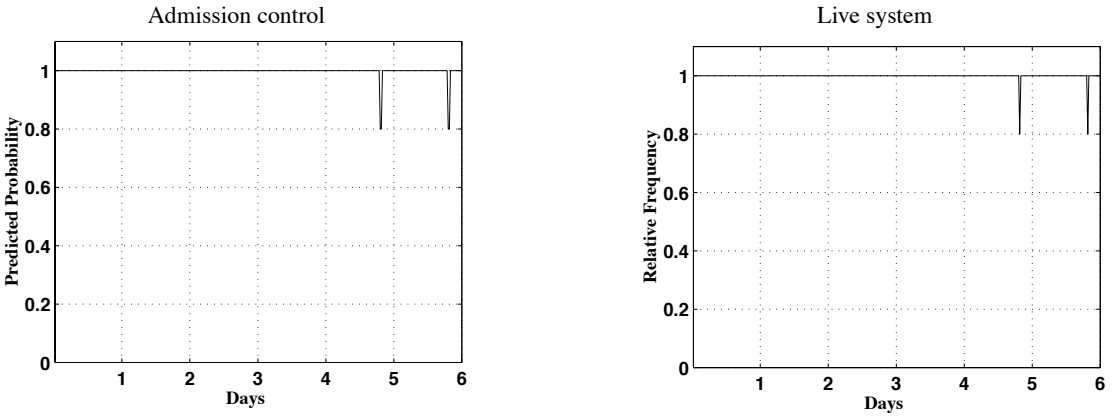

Figure 4: **CPU:** $target_{WC}$ **= 200,** $target_{IBM}$ **= 450,** $target_{Rice}$ **= 24**

We next consider admitting the service corresponding to the Rice 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 Rice, 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.

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

sponds to a 24 minute interval of trace time

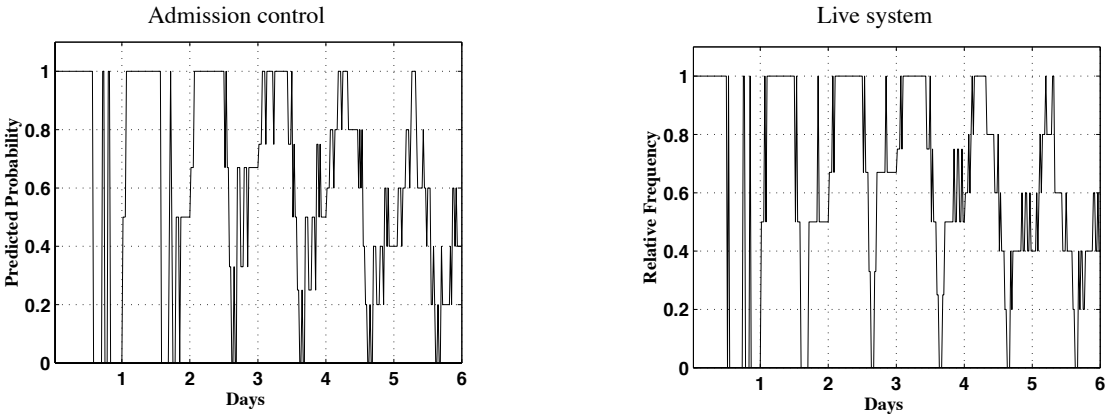

Figure 5: **CPU:** $target_{Rice}$ **= 100,** $target_{WC}$ **= 250,** $target_{Google}$ **= 50**

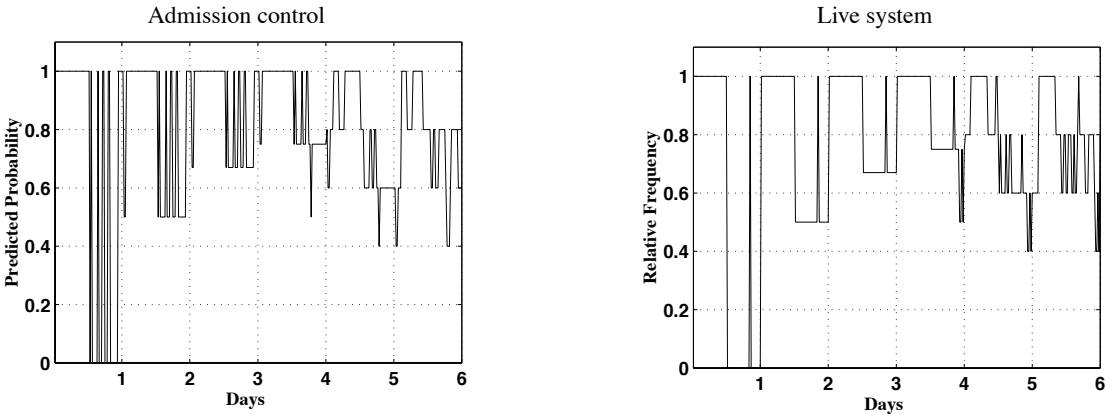

Figure 6: **Disk:** $target_{Rice}$ **= 100,** $target_{WC}$ **= 250,** $target_{Google}$ **= 50**

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

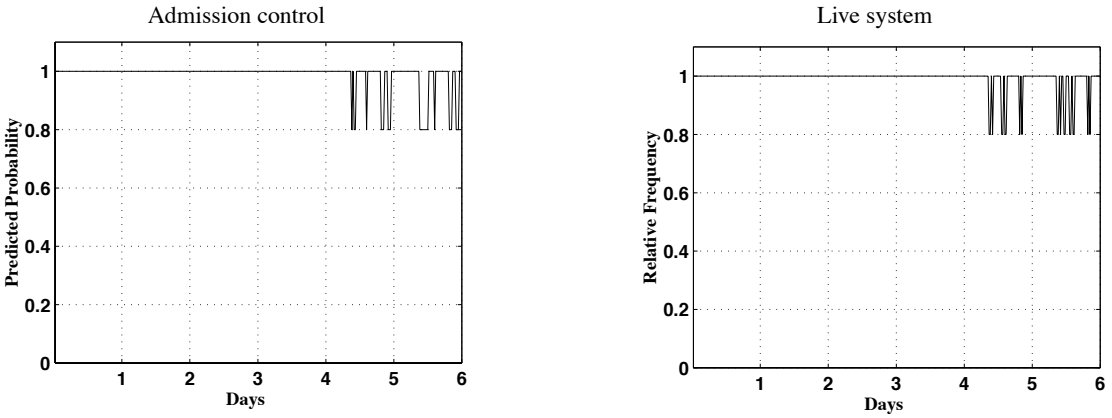

Figure 7: **CPU:** $target_{Rice}$ **= 100,** $target_{WC}$ **= 250,** $target_{Google}$ **= 5**

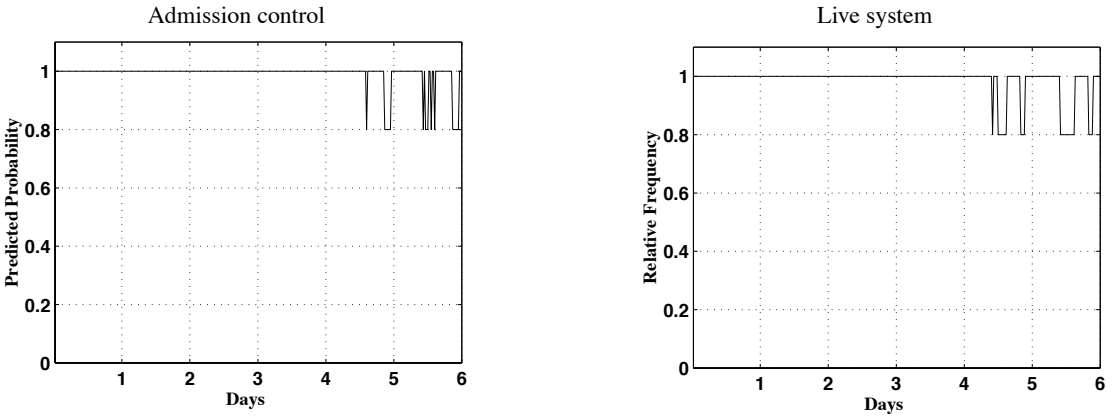

Figure 8: **Disk:** $target_{Rice}$ **= 100,** $target_{WC}$ **= 250,** $target_{Google}$ **= 5**

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

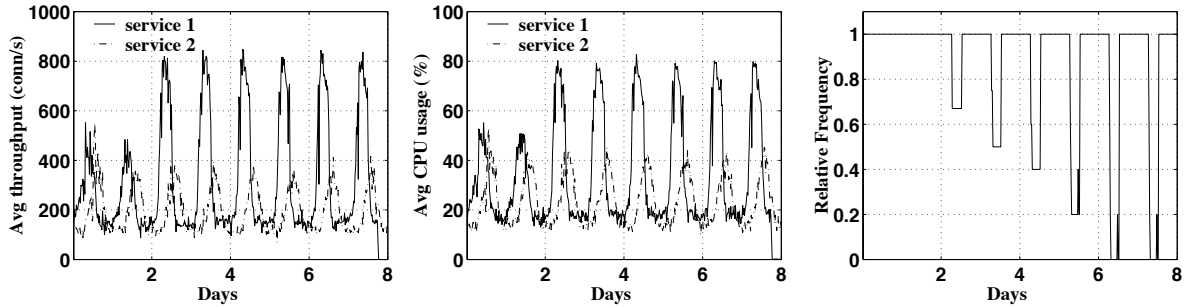

Figure 9: **CPU:** $target_1$ **= 1000,** $target_2$ **= 500**

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.

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

This section describes background material in the areas of resource management and scheduling. Then it contrasts against some recent work in applications of OS-level resource management to web server QoS. Finally it describes some parallels in the domains of application-based web server QoS and also network QoS.

6.1 Resource management and scheduling

The framework presented in this paper builds on prior work in mechanisms for performance isolation in operating systems [6, 10, 11, 21, 33] and in proportional share scheduling policies for CPU and disk bandwidth [9, 18]. Mechanisms like Resource Containers [6] separate the notion of resource principals from processes, thereby allowing fine-grained resource accounting and scheduling for individual services in Web server-like applications. 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. Mechanisms such as Cluster Reserves [4] extends this concept to resource principals on a cluster-wide scale. This paper derives from prior work, by presenting a complete framework for the provision of *predictable services*.

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 [14], resource containers are capable of affording performance isolation on a single node. Similar ideas for overload protection and service differentiation were presented by Voigt et al. [35]. Other related abstractions are Activities [21] in the Rialto real-time operating system, software performance units (SPU) [33] proposed in the context of shared-memory multiprocessors, reservation domains in the Eclipse operating system [11, 10] and paths in Scout [29]. 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 [36] and STFQ [18]. The SMART [24] multimedia scheduler integrates

priorities and weighted fair queuing to meet real-time constraints while simultaneously supporting non real-time applications.

Steere et al. [30] describe a feedback-based real-time scheduler that provides reservations to applications based on dynamic feedback, eliminating the need to reserve resources *a priori*. In contrast, our feedback-based scheduler augments reservation-based schedulers to achieve performance targets using application feedback. Thus, an important conceptual contribution of our scheduler is that of realistically bridging the gap between application-level QoS metrics, i.e. those of immediate interest to clients, and kernel-level notions of resource utilizations and needs.

6.2 QoS through OS-level resource management

Concurrent to our work, Urgaonkar et al. [32] have proposed a system to provide a different type of server QoS, viz. to perform *resource overbooking* while hosting services on shared server platforms. The authors first derive the resource usage envelopes of services in a manner similar to ours, using OS-level resource monitoring on an isolated node running the service. Then their key idea is to allocate resources to services corresponding to a high percentile of the resources needed to satisfy all requests. This allows the server node to be significantly overbooked (thus increasing utilization), while incurring a small degradation in performance, i.e., a small fraction of requests experiencing long delays or being dropped.

Our work differs from the above in a number of ways; the following list describes our contributions. In general, we take a more systemic and practical-minded approach, and solve some different aspects of the larger problem.

1. [32] implicitly treats applications' resource needs as that required to service all incident requests (and then allocates a high percentile of these needs). Instead, our system allows applications to define their own performance metric, with a target value that describes the resource need. This *flexibility* allows services to, for example, define the metric as throughput or the reciprocal of response time or interesting combinations of the two metrics, thus catering to the needs of diverse application types (such as streaming media, gaming servers, etc). Another advantage is that our approach is more amenable to a *mixture* of different types of services, with different performance metrics, co-located on a server node.
2. Our work is the first to address and demonstrate the management of *multiple resource classes* (CPU, disk and memory) in a shared hosting situation. In Section 3.4, we describe and solve three novel problems that arise due to the idiosyncrasies of some resource classes and the interactions between them. Our experiments were not network bandwidth intensive. [32] has provisioned CPU time and network bandwidth between services using separate schedulers.
3. Our system recognizes the daily and weekly *periodic nature* of common web workloads (see web logs depicted in the Appendix), and takes advantage of the statistical multiplexing opportunity when sites catering to different timezones peak at different times. We therefore use time-dependent values of the service's resource usage and QoS target, whereas [32] computes the time-independent distribution of resource usage.
4. We propose a *resource monitor* to react to long-term load variations, and evaluate it in Section 5.3. [32] only measures resource usages when the service is under trial; it does not describe any continuous monitoring facility or reaction mechanism.
5. While [32] evaluates their system on synthetic benchmarks (e.g., static and dynamic SPECweb99), we perform *trace-based experiments*, including a CGI workload based on the Google trace. One of the consequences is that we exercise resources such as disk and memory in different ways.
6. Our implementation is centered around a single machine, whereas [32] also addresses the problem of shared hosting on a cluster of server nodes, i.e., the *capsule placement problem*. We believe that our framework can be similarly extended to a cluster, partly using a cluster-wide resource management and load distribution mechanism such as Cluster Reserves [4]. Exploring this aspect further is the subject of future work.

6.3 QoS through application-level scheduling

This paper and [32] use OS-level resource scheduling to make admission control decisions and meet QoS contracts. One may try to achieve the same effect by simply scheduling requests enroute to the respective services. Though such an approach is simpler, it would be encumbered by insufficient information about resource needs, and can only use request scheduling to exercise indirect control on resource allocations. Hence application-level request scheduling is used primarily for *service differentiation*; three items of related work are described here.

Li and Jamin [22] 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 [7] 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.

Shen et al. [27] define a performance metric called *quality-aware service yield* as a composition of throughput, response time, etc., and propose application-level request scheduling algorithms (at the cluster and within each node) to optimize the yield criterion. They use this system to enable service differentiation in the Ask Jeeves search engine. Their main focus is to compare scheduling policies; they do not perform admission control on incoming services, or provide guarantees on target values of the performance metric. While they acknowledge the desirability of wanting to co-locate CPU-intensive services with IO-intensive ones, they only consider the CPU resource in their implementation, and leave multi-dimensional resource accounting for future work.

6.4 Network QoS

Support for scalable service differentiation has been provided in the Internet by means of a special field in the IP packet headers [23, 8]. This field is set at network boundaries and determines how packets are forwarded by nodes inside a network. Scalability is achieved by eliminating the need for maintaining per-flow state and signalling at every hop.

Jamin et al. [20] describe a measurement-based admission control algorithm for predictive services in packet networks. The algorithm models the traffic of a predictive service using a token bucket that is parameterized using measured bandwidth utilization and packet queuing delay for that service. A new service is admitted only if the bandwidth requirements can be met and if the delay bounds for existing services are not violated. The algorithm is shown to achieve high network utilization while permitting occasional delay violations. 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.

7 Concluding remarks

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 levels of quality of service.

The framework consists of a measurement-based admission control procedure 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, its desired quality of service level, and its 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.

Conceptually, the system closes the gap between application-oriented QoS metrics, i.e. those of immediate interest to content providers, and kernel-level notions of resource usage that the site operator

can schedule and manage. The system encompasses all resource classes on a machine, and solves several interesting and complex issues in scheduling multiple resource classes simultaneously.

An 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. Thus, the prototype gives a proof of concept for effective, predictable quality of service and capacity planning in Web servers and proxies.

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

[8] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC 2475: An Architecture for Differentiated Services, Dec. 1998.

[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. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[14] 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.

[15] 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.

[16] Ensim. http://www.ensim.com/.

[17] 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.

[18] 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.

[19] S. M. Hand. Self-Paging in the Nemesis Operating System. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[20] 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.

[21] 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.

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

[23] K. Nichols, S. Blake, F. Baker, and D. Black. RFC 2474: Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers, Dec. 1998.

[24] 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.

[25] OneSquared. http://www.onesquared.net/.

[26] 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.

[27] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[28] Site5 Internet Solutions, Inc. http://www.site5.com/.

[29] 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.

[30] D. C. 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 3rd USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[31] 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.

[32] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[33] 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.

[34] Verizon Online. http://biz.verizon.net/web/hosting/.

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

[36] 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

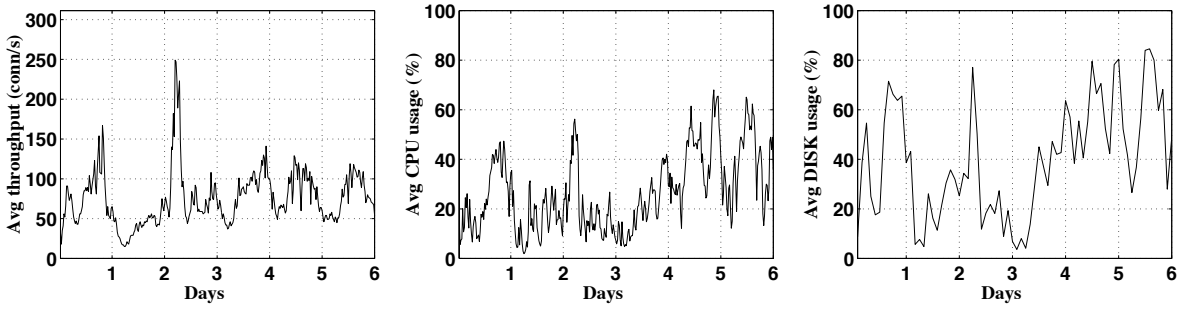

Figure 10: **Rice University trace**

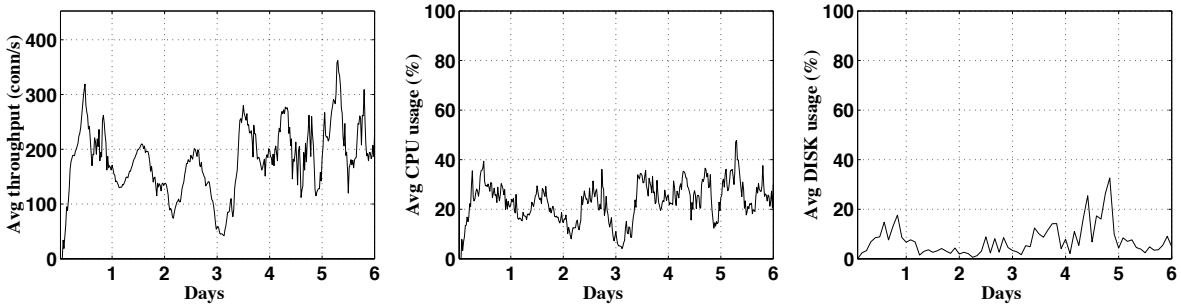

Figure 11: **1998 Soccer World Cup trace**

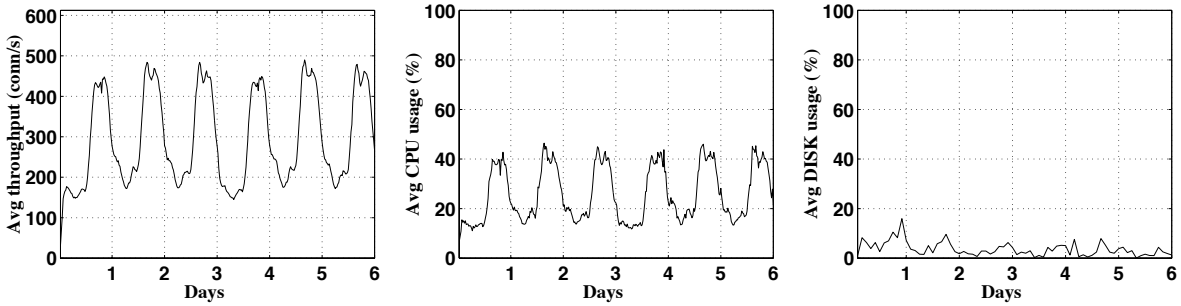

Figure 12: **IBM trace**

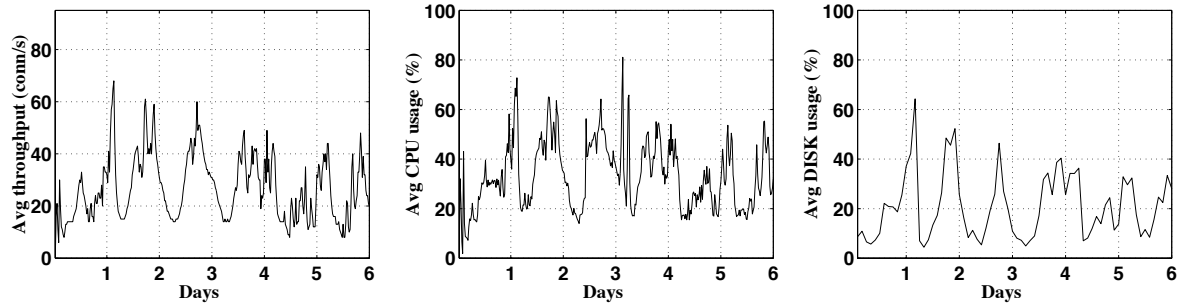

Figure 13: **Google trace**

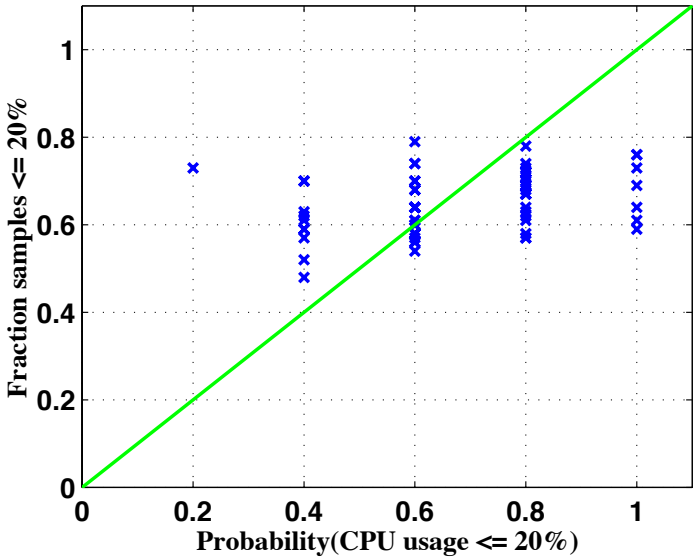

Figure 14: **Accuracy of future extrapolations**