The Impact of Concurrency Gains on the Analysis and Control of Multi-threaded Internet Services

Hani Jamjoom Chun-Ting Chou Kang G. Shin

University of Michigan

{jamjoom,choujt,kgshin}@eecs.umich.edu

Abstract— **With the proliferation of Internet services, many solutions have emerged to provide Quality-of-Service (QoS) guarantees when the demands for the hosted services exceed the server's capacity. In this paper, we take an analytical approach to answering key questions in the design and performance of application-level QoS techniques, especially those that are based on the multi-threading or multi-processing abstraction. Key to our analysis is the integration of the effects of concurrency into the interactions between multi-threaded services. To this end, we extend traditional time-sharing models to develop the multi-threaded round-robin (MTRR) servers, a more accurate model of operation of typical multi-threaded Internet services. For this model, we first develop powerful, yet computationally-efficient, mathematical relationships that describe the performance (in terms of throughput and response time) of multi-threaded services. We then apply optimization techniques to derive the optimal allocation of threads given specific QoS objective functions. Using realistic workloads on a typical web server, we show the efficacy and accuracy of the proposed new methodology.**

I. INTRODUCTION

Wide use and expansion of the Internet has led to the proliferation of diverse and oftentimes complex Internet services. These services, on the other hand, have created unprecedented demands on end-servers, each of which usually hosts multiple services like Web, e-mail, and database services. The increased demands by end-users often outpace the recent progress in enhancing server's processing, storage and networking capacities, hence easily overloading end-servers. The notion of Quality-of-Service (QoS) has been introduced to manage resources when user demands exceed resource supplies. Supporting QoS in servers has been addressed extensively in the literature, for example, in [2, 4, 6, 11, 32]. In particular, application-level QoS mechanisms are designed to provide the necessary QoS guarantees with little or no support from the end-server's OS [2, 10, 13, 15, 22, 25]. However, since the underlying OS enforces resource transparency (i.e., hides resource management), application-level mechanisms have limited capabilities in enforcing strict service guarantees and are often restricted to only providing proportional QoS differentiation. In this paper, we closely examine and evaluate the extent to which application-level mechanisms can provide QoS support.

The work reported in this paper was supported in part by the National Science Foundation under Grant CCR-0216977.

One of the more popular application-level solutions is *thread-based QoS mechanisms* [23, 24, 32] in which the allocation of threads or processes to each application or service is adjusted (either statically or dynamically) based on some target QoS objectives (Figure 1). Two design principles motivate the use of thread/process allocation to provide QoS differentiation to multiple services: (1) increasing concurrency improves the performance of a single service, and (2) server capacity can be divided in proportion to the thread allocation. Unfortunately, the extent to which thread-based mechanisms are effective depends heavily on the degree of interaction between the running threads, which further depends on the nature of the workload of incoming requests. This paper carefully examines each of the two design principles with the goal of providing deeper understanding of internal dynamics behind this QoS mechanism.

When a service is allocated more threads, the advantages of increased concurrency are apparent in the resulting increase in throughput. This improvement is due to concurrent processing of requests, which allows the overlapping of long blocking I/O operations of one request with non-blocking operations of another. There is, however, a *saturation point* beyond which increased concurrency no longer yields any performance benefits. When multiple services use concurrency to improve their own performance, the interaction between their threads become more complex. In fact, we have found that as the system's load increases, the performance interaction between different service classes, due to resource sharing, becomes less predictable. Furthermore, when different types of workloads (e.g., I/O-heavy and CPU-heavy) are sharing the system, a marginal improvement in the QoS of one service can cause a dramatic decrease in the QoS of the other services. Based on our measurements and observations, we show that multi-threading is ill-suited for providing application-level QoS support. On the other hand, it can be effectively used to provide QoS guarantees to different client groups.

In this paper, we take an analytical approach to precisely characterize the interactions between threads and services in an Internet server. Crucial to the correctness of our analysis is the development of an accurate model that reflects the operation of the server. We introduce the multi-threaded

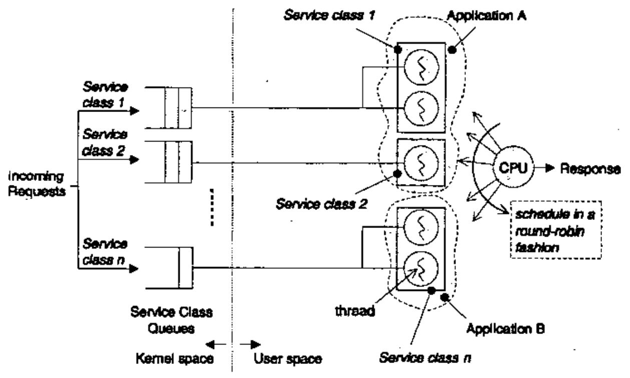


Fig. 1. Thread-based system model. Application A uses two service classes to give preferential treatment for requests in service class 1 than requests in service class 2. Application B uses one service class to enforce a certain QoS to all incoming requests. A controller (not shown) can then adjust the allocation of system threads to different service classes.

round-robin (MTRR) server model to capture the multi-threading and process-sharing abstractions of real systems. The MTRR model is an extension of traditional round-robin servers, which are used in the analysis of polling and time-shared systems [19, 20, 29]. Unlike traditional approaches, our model incorporates the performance benefits of increased concurrency into the interaction between the running threads. Using this MTRR model, we are able to derive powerful, yet efficient, relationships that describe the internal dynamics of a typical multi-threaded server. Furthermore, these relationships allow us to address three important issues in the design and performance of application-level QoS differentiation: (1) better predict the impact of thread concurrency on client-perceived delay than traditional models, (2) estimate the expected performance of services for any thread allocation, (3) find the thread allocation, if any, that guarantees certain response times to different client groups (e.g., paying customers are given preferential treatment over the non-paying ones).

This paper is organized as follows. We analyze, in Section II, the benefits of concurrency in multi-threaded applications. We then establish the server and application models for our analysis in Section III. Section IV presents a detailed analysis of the MTRR server to provide the basic relationships governing the performance of multi-threaded services. In Section V, we look at the effects of workload dependencies on the analysis of multiple services being hosted on a single server. We then provide, in Section VI, a computationally-efficient algorithm for determining the optimal allocation that meets various QoS objectives. We use real measurements on a typical Web server in Section VII to evaluate the correctness our derivations and effectiveness of our allocation algorithm. We review related work in Section VIII. Finally, in Section IX we conclude the paper with our final remarks.

II. QUANTIFYING CONCURRENCY GAINS

Using concurrency to improve server performance is one of the guiding principles for providing thread-based QoS support. This notion was explored in [23, 24, 32] as an integral part of their feedback control mechanism that increases the allocation of threads to running applications when better performance is needed. Implicit to the effective operation of these mechanisms is the notion that increasing the number of threads improves the performance of the application. Particularly, the performance gain due to increased concurrency is normally split into three regions as shown in Figure 2: (I) a linear increase region due to overlapping blocking operations of some threads with non-blocking operations of the other threads, (II) flat or no-gain region due to threads contending for the bottleneck resource, and (III) sudden (or exponential) drop region due to memory thrashing. In this section, we establish this behavior for different workloads on a real system. This will set the stage for exploring the impact of concurrency on the controllability of multi-threaded applications.

We define $G_k(m)$ as the *speedup (or gain) function* that expresses the potential performance gain (or loss) when m threads run concurrently. Because the expected speedup is workload-dependent, the function needs to be profiled for each specific workload, denoted by the subscript k. The speedup function expresses the change in throughput rather than the change in response time. This is because increasing concurrency does not reduce the actual amount of work that each request needs. Instead, it increases the efficiency of the server, which can be captured by the improved throughput. To profile $G_k(m)$, we first measured the maximum service throughput, $\hat{\mu}_k(m)$, when m threads run concurrently. This is done by limiting the application to have a maximum of m concurrent threads (for $m = 1, 2, \ldots$) and configuring the arrival rate to be high enough to keep all threads busy processing incoming requests. The speedup function is, then, the throughput gain when m threads are allocated compared to when a single thread is allocated. Specifically,

$$G_k(m) = \frac{\hat{\mu}_k(m)}{\hat{\mu}_k(1)}.$$

To illustrate the general characteristics of concurrency improvements, we configured a server machine (a 2.24 GHz Pentium 4 with 1 GBytes of RDRAM) to run Apache 1.3 and receive HTTP requests through a high-speed FastEthernet link. Three Linux-based machines are used to generate the desired requests. Our load generator, Eve [17], follows the same design principles provided by SPECWeb99 [12], a widely-used tool to evaluate the performance of Web servers, to test static and dynamic workloads.[1] The primary difference

[1]The static workload consists of only static objects, resembling web pages and embedded images. The dynamic workload is similar to the static one, except the requested objects are created on-the-fly for each incoming request using CGI scripts.

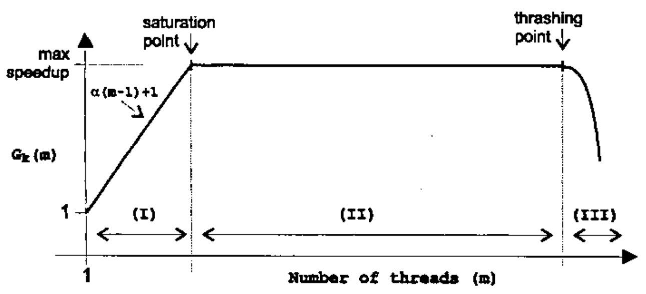


Fig. 2. Shape of the speedup function, $G_k(m)$. The function increases at a linear rate in region I up to the saturation point. The function, then, flattens out in region II and suddenly drops after the collapse point in region III.

between the two load generators lies in our ability to sustain an arrival rate regardless of the progress of on-going requests. In contrast, SPECWeb99 sends a fixed maximum number of requests; once the maximum is reached, a new request is sent only after the completion of a previous one. We profiled $G_k(m)$ for three workloads: purely static, purely dynamic, and mixture of the two — mixed for short. Each workload adheres to the specification provided by SPECWeb99; in general, the requested files follow a Zipf distribution [8] regardless of whether they are statically or dynamically generated.

Figure 3 shows $G_k(m)$ for the three workloads, with the abscissa drawn in log-scale. The first two regions outlined earlier are clearly depicted by the figure, where the linear region is reflected by the sub-linear growth in the log scale. The combination of having a fast machine with large memory and running processes with small memory footprints prevented reaching the collapse point. This was the case even when a very large number of processes run simultaneously.

The width of the linear increase region (i.e., region I) in Figure 3 and its slope depend heavily on the type of workload. We approximate the speedup function in region I by a simple linear function:

$$G_k(m) = \alpha_k(m-1) + 1 \qquad \text{for } m = 1, \ldots, m_k^t,$$

where m_k^t reflects the saturation point (defined later), and the slope, α_k, reflects the *speedup rate*, or alternatively, the efficiency of concurrency for workload k. In the ideal case, where each additional thread behaves as an independent server, $\alpha_k = 1$. This is seldom the case, and therefore, $\alpha_k \leq 1$. The mixed workload, for instance, had a speedup rate $\alpha_{mix} \approx 0.14$ and a linear increase region of $m^t \approx 23$ threads. If the workload is purely CPU-based or purely I/O-based, then one expects little performance gain since blocking and non-blocking operations are not overlapped. In that case, $\alpha_k = 0$.

The transition point between regions I and II, which we call the *saturation point* (m^t), is primarily due to threads contending for the bottleneck resource — usually the disk. When a single class is being controlled, increasing the number of threads to be allocated beyond the saturation point provides

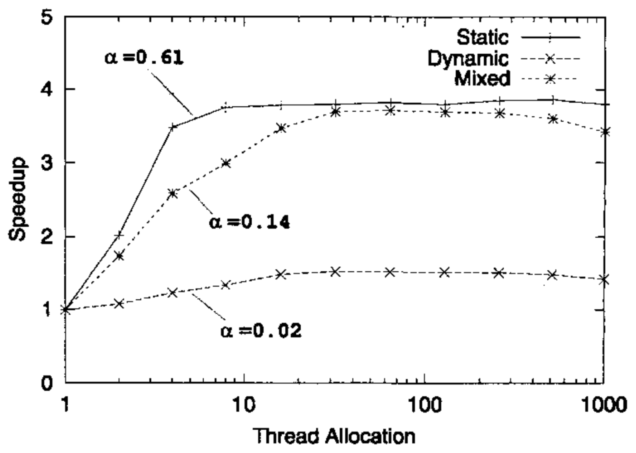


Fig. 3. Speedup function, $G_k(m)$, for static, dynamic, and mixed workloads.

no performance advantage to the hosted service. But when multiple services are being controlled, recognizing the saturation point becomes more crucial since adding more threads to one service class reduces the second class' share of the system. This may cause the second class to increase its thread allocation and create a vicious cycle between the two classes rendering the underlying control mechanism ineffective. It is thus necessary for any dynamic control mechanism to adjust the maximum thread allocation based on the observed throughput; when no throughput gain is observed, then no further threads should be allocated. This issue is explored closely in the remainder of the paper.

III. MODELING MULTI-THREADED SERVICES

The complexity of today's servers presents a real challenge in building analytical models that fully describe the dynamics of the underlying server. Our goal is, thus, to create a model that is simple enough to allow for mathematical tractability, yet accurate enough to reflect realism. Specifically, the created model must capture the effects of concurrency as well as the basic interaction between the various running threads. In this section, we give a detailed specification of our system by describing the computing model, which details the assumed operation of a typical multi-threaded server, and the workload model, which specifies the arrival and service-time distribution of incoming requests.

A. Computing Model

Our computing model is based on a general understanding of the typical operation of current time-sharing OSs and Internet services. We use an MTRR server to model a general computing environment where a single processor is shared by multiple threads.[2] Threads are assumed to be the smallest allocatable unit of work and are distributed among n service

[2]We use the terms "threads" and "processes" interchangeably throughout this paper.

classes, $\{S_1, S_2, \ldots, S_n\}$. Specifically, each service class S_k is allocated m_k^0 threads and has an independent buffer of size B_k to hold the requests that cannot be processed immediately. We use the term "service classes" as opposed to just "services" to capture the situation where a single service is configured to differentiate between multiple client populations (Figure 1). An example of this is Apache's *Virtual Host (VH)*, where, for instance, clients from network `192.168.10.x` are serviced using one VH and clients from the remaining IP address-space are serviced using another VH. Thus, our Apache service is said to have two service classes. In contrast, if an application does not differentiate between clients, the entire application is represented by a single service class. Using the notion of a service class, therefore, allows us to capture QoS differentiation between different applications and also between client groups within a single application.

Beside having threads as a shared resource, dependencies between service classes arise due to two possible interactions: (1) they share a bottleneck resource such as a disk and (2) they are organized as a series of stages where an incoming request must be processed by multiple services in a particular order [10, 32]. The complexities that are introduced by the latter is akin to those in network of queues [9], but with dependent service distributions. In this paper, we restrict our analysis to single-stage services and focus on the dependencies due to resource sharing. We, thus, make the following assumptions for the internal operation of an MTRR server.

A1. A request is assigned to a working thread. Multiple requests can be processed simultaneously by running multiple threads and time-sharing the system. We assume that all threads are homogeneous,[3] even though they can be assigned to different service classes. This is in line with actual OS operation as system threads can be created and removed easily with little overhead.

A2. A thread is either running, ready, or blocked waiting for a new incoming request. Basically, a ready thread is waiting for its share of the server to continue processing a request, and a blocked thread is waiting for a new request. We do not consider alternate states in which a thread is waiting for other operations to complete such as blocking for I/O. These are captured by the speedup function.

A3. All threads are of equal priority. Service priorities have been studied in both queueing and real-time systems [21, 33]. Including service priorities in our model will, unfortunately, complicate our analysis and is, thus, omitted from our model.

A4. Threads (in the ready state) are scheduled (by the underlying OS) in a round-robin fashion, each for Q seconds or until the thread finishes processing the current request, whichever happens first. The task of servicing all ready threads *once* is called a *service round*. We do not consider the effects of hierarchical priority queueing, which is commonly used to age long-running threads. Since all requests are relatively short-lived and all threads have the same priority, a strict round-robin algorithm can be assumed.

A5. Switching between different running threads is done instantaneously with no overhead. Similar to A2, we capture this overhead in the speedup function, and hence, this is not a limitation. Our decision is motivated by the fact that switching overhead is load-dependent. That is, as more threads are running, switching between threads will depend on whether the threads need to be swapped out of memory or not. The speedup function allows us to include load-dependent overheads in our analysis.

A6. The system has a fixed (finite) number of threads, m^{max}. This corresponds to the maximum number of threads that a typical OS can support. Not all threads need to be allocated, but, the total number of threads that are allocated to all service classes cannot exceed this limit.

One final point to make is that our analysis does not consider any particular server resource as the bottleneck resource. Instead, the server is limited by the rate at which it can process requests and this rate is defined by the service-time distribution and speedup function of incoming requests.

B. Workload Model

In an Internet server, the workload model captures the arrival of requests and service that each request requires. Both have been studied extensively in the literature [3, 5, 7, 14, 27]. In general, they have been observed to follow heavy-tail distributions. In fact, we have observed similar behavior during our workload analysis (omitted for space considerations). Heavy-tail distributions are, unfortunately, difficult to analyze even with very simple computing models. In order to provide better understanding of the dynamics of multi-threaded services, we assume that requests arrive following a Poisson process and require exponential service times. Section VII evaluates our model using realistic load distributions.

We distinguish between *service time* and *processing time* of an incoming request. The former reflects how much work that each request brings to the system, whereas the latter reflects how much time it spends in service as it shares the system's resources with other requests. Thus, there are three parameters associated with each service class S_k:

λ_k:	the mean request arrival rate of a Poisson arrival process.
$1/\mu_k$:	the mean service time of each request. It is equal to the processing time only when the system is allocated a single thread.
$G_k(m)$:	the speedup function as defined in Section II. Even though we use the subscript k to denote the service class, not the workload, the characterization of $G_k(m)$ remains unchanged. For example, if $G_1 = G_2 = G_{static}$, it implies that both service classes have static workloads.

[3]That is, we do not mix different types of threads such as application-level and kernel-level threads.

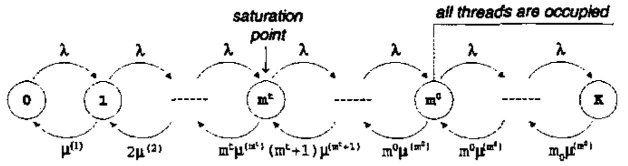


Fig. 4. Markov Chain representation of MTRR server.

We assume that $G_k(m)$ only operates in regions I and II (Figure 2). The point where $G_k(m)$ collapses is hard to predict *a priori*, but not very difficult to detect [26, 31, 32]. For the purpose of our analysis, we assume that detection/prevention from server thrashing is handled by a separate mechanism. Therefore, we define the speedup function as follows:

$$G_k(m) = \begin{cases} \alpha_k(m-1)+1 & \text{for } 1 \le m \le m_k^t \\ \alpha_k(m_k^t-1)+1 & \text{for } m > m_k^t, \end{cases} \quad (1)$$

where α_k is the constant reflecting the efficiency of concurrency and m_k^t is the saturation point of service class k.

IV. ANALYSIS OF MTRR SERVER

We focus in this section on the single service class MTRR server. The analysis, however, requires extension of some of the existing results from queueing theory and time-sharing systems [19, 20, 33] to include the effects of concurrency gains. This is done by introducing state-dependent service rates through the speedup function. We first consider an idealized model where the scheduling quantum is infinitesimal, i.e., $Q \to 0$. Under this assumption, we are able to model the MTRR server using Continuous-Time Markov Chain (CTMC) [33]. Later we will estimate the resulting error from this assumption.

Figure 4 shows the basic representation of the CTMC of the single-class MTRR server. The state here represents the number of requests in the system, and $\mu^{(i)}$ represents the state-dependent service rate, not the per-service class parameter, μ_k, described earlier. Therefore,

$$\mu^{(i)} = \frac{\mu}{i}G(i), \quad (2)$$

where μ and $G(i)$ are the parameters describing the single service class under study. We drop the subscript k as there is only one service class.

We start by writing the steady-state probabilities for the CTMC, which are based on the local balance equations:

$$p_i = \begin{cases} \frac{\lambda}{i\mu^{(i)}}p_{i-1} & \text{for } i = 1, \ldots, m^0 \\ \frac{\lambda}{m^0\mu^{(m^0)}}p_{i-1} & \text{for } i = m^0+1, \ldots, K. \end{cases} \quad (3)$$

where m^0 is the number of allocated threads, and K is the maximum number of requests that can be admitted into the system, which includes requests both in queue and in service. Specifically, $K = B + m^0$. Using Eqs. (1) and (2), we rewrite the expressions for probabilities as:

$$p_i = \begin{cases} \frac{\lambda}{[(i-1)\alpha+1]\mu}p_{i-1} & \text{for } i = 1, \ldots, \hat{m} \\ \frac{\lambda}{[(\hat{m}-1)\alpha+1]\mu}p_{i-1} & \text{for } i = \hat{m}+1, \ldots, K. \end{cases} \quad (4)$$

where α represents the speedup rate as described in Section II and $\hat{m} = min(m^t, m^0)$. Notice the change of indicies in Eq. (4) from m^0 to $\hat{m}$ since $\mu^{(i)}$ remains unchanged for $i \ge \hat{m}$. Let us define $\rho = \frac{\lambda}{\mu}$ and also $\Psi_\alpha(i)$ as follows:

$$\Psi_\alpha(i) = \begin{cases} 1 & \text{for } i = 0 \\ \prod_{k=1}^{i}(k\alpha+1) & \text{otherwise.} \end{cases} \quad (5)$$

Now using simple substitution, we can rewrite the expression of each p_i as a function of p_0.

$$p_i = \begin{cases} \frac{\rho^i}{\Psi_\alpha(i-1)}p_0 & \text{for } i = 1, \ldots, \hat{m} \\ \frac{C^{\hat{m}}}{\Psi_\alpha(\hat{m}-1)}\left(\frac{\rho}{C}\right)^i p_0 & \text{for } i = \hat{m}+1, \ldots, K. \end{cases} \quad (6)$$

where $C = (\hat{m}-1)\alpha+1$. Since $\sum_{i=0}^{K} p_i = 1$, we can express p_0 as follows:

$$p_0 = \left[1 + \sum_{i=1}^{\hat{m}} \frac{\rho^i}{\Psi_\alpha(i-1)} + \frac{\rho^{\hat{m}} - \frac{\rho^K}{C^{K-\hat{m}}}}{\Psi_\alpha(\hat{m}-1)(\frac{C}{\rho}-1)}\right]^{-1}. \quad (7)$$

Using these steady-state probabilities, one can numerically compute the expected number of requests in the system, $\overline{N} = \sum_{i=0}^{K} i.p_i$. Little's formula [33], $\overline{N} = \lambda(1-p_K)\overline{W}$, can then be used to compute the total response time, $\overline{W}$, which includes both the queueing and processing delays. The term $(1-p_K)$ is used to account for the probability that an arriving request finds a full queue and thus is dropped.

Given specific values for the system parameters, computing the various results is straightforward and can be achieved in $O(K)$ operations. Our formulation of the MTRR server along with the introduction of the speedup function constitutes a superset of several well-studied systems. For instance, when $\alpha = 0$, we observe no speedup. This reduces to a Generalized Processor Sharing (GPS) without priorities [18]. If we further add the restriction of $m^0 = 1$, then only a single thread is allowed to run. The system is further reduced to $M/M/1/B$ server. Finally, if $\alpha = 1$, it implies ideal speedup or, effectively, m^0 servers running in parallel. The system then becomes $M/M/m^0/B$.

subsectionComparison with Discrete Quantum Values

The development so far assumed an idealized case of $Q \to 0$. Here we want to give a general idea of the expected error that is introduced by this assumption. For simplicity, we consider the worst-case scenario where the service is heavily-loaded, i.e., m is always equal to m^0. We also assume no speedup, i.e., $G(m) = 1$. When $Q \to 0$, the mean processing time, $\overline{Y}$, is just

$$\overline{Y} = \frac{m^0}{\mu}. \quad (8)$$

Now, let Q be a positive real value — typical values are

0.01 sec. We want to derive an approximate expression for $\overline{Y}$. We consider the processing of a request by one of the m^0 threads that always run during any service round (assumption A4). Let X be an exponential random variable reflecting the service time of an arriving request. Upon admission of the request into service at the beginning of a service round, its corresponding thread must first wait for its service turn before it starts execution. When the thread is scheduled, if it completes servicing the request in less than a time quantum (i.e., $X \leq Q$), its processing time is just the sum of X and the queueing delay before it starts service. On the other hand, if $X > Q$, then we expect that after Q seconds, the remaining threads must run before the beginning of the next service round, where the given thread must wait for its turn to run again. This process repeats until the request is completed.

The time that a thread must wait for m other threads to be serviced, either before or after it is scheduled, can be computed as follows:

$$\begin{aligned} E[V|m] &= E[\sum_{i=1}^{m} min(X,Q)] = m.E[min(X,Q)] \\ &= m\left(\int_0^Q x f_X(x)dx + \int_Q^\infty Q f_X(x)dx\right) \\ &= m\frac{1-e^{-\mu Q}}{\mu}. \end{aligned} \quad (9)$$

where $f_X(x)$ is the probability density function (pdf) of X.

During each service round, we assume that the order of scheduling threads is completely random. That is, for any given thread, its probability of being scheduled at the k-th position is $1/m^0$. We can now compute $\overline{Y}$ using the so-called *regenerative formulation*:

$$\begin{aligned} \overline{Y} &= \sum_{k=0}^{m^0-1} \frac{1}{m^0}\Big\{E[V|k] + \int_0^Q x f_X(x)dx \\ &\quad + \int_Q^\infty \left(E[V|m^0-k-1] + \overline{Y} + Q\right) f_X(x)dx\Big\} \\ &= \frac{1}{\mu}\left[\frac{m^0+1}{2} + \frac{m^0-1}{2}e^{-\mu Q}\right]. \end{aligned} \quad (10)$$

When $Q \to 0$ in Eq. (10), we see that the results are consistent with Eq. (8). Furthermore, the error between the two equations is

$$\begin{aligned} \%Error &= \frac{\frac{m^0}{\mu} - \frac{1}{\mu}\left[\frac{m^0+1}{2} + e^{-\mu Q}\frac{m^0-1}{2}\right]}{\frac{m^0}{\mu}} \\ &= \frac{m^0-1}{2m^0}\left[1-e^{-\mu Q}\right]. \end{aligned} \quad (11)$$

In the case of the mixed workload, where $\mu = 50$ reqs/s, the expected error is approximately 19%. We stress, however, that this is a worse-case scenario. In our experiments, we found that our derivations are within 10% of real measurements for a wide range of configuration parameters.

We note that while using finite Q values to determine $\overline{Y}$ better approximates the real system behavior, it is mathematically tractable when $G(m) = 1$. When $G(m) > 1$, this method incorrectly reduces the processing times as it underestimates the number of service rounds required for completing a single request.

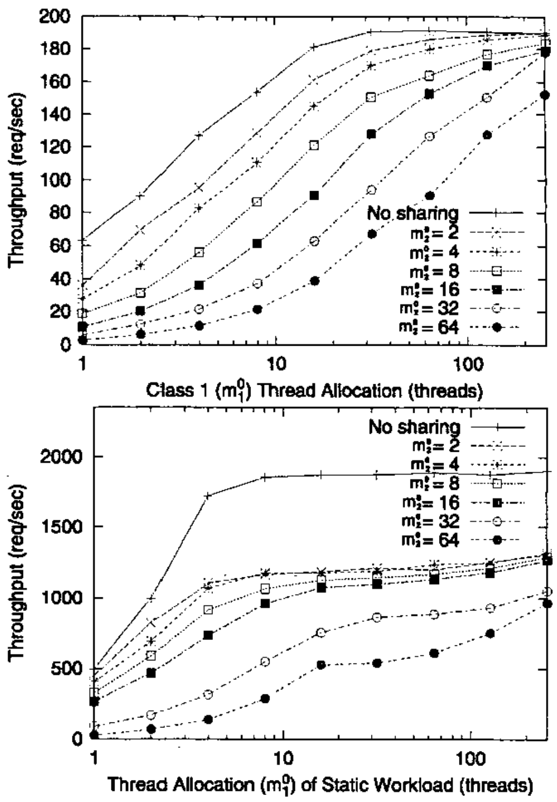


Fig. 5. Workload interdependencies. (top) homogeneous workloads, (bottom) heterogeneous workloads.

V. WORKLOAD DEPENDENCIES

When multiple service classes run on the same host, their basic operation is similar to an MTRR server with a single service class. But instead of allocating all m_0 threads to one service class, there are n service classes, $\{S_1, S_2, \ldots, S_n\}$, and each service class S_i is allocated a fixed number of threads m_i^0 such that their sum does not exceed the system-wide thread limit m^{max} (Assumption A6). Each service class also has separate workload parameters: $(\lambda_i, \mu_i, G_i(m))$. With the introduction of multiple services into our model, the analysis must consider two interdependencies between service classes. The first, which we call *direct interdependencies*, is due to threads time sharing the system. The second, which is *indirect interdependencies*, is caused by the possible sharing of a bottleneck resource such as the disk. We examine how these interdependencies affect the analysis, and hence the performance, of servers running multiple services.

The distinction between direct and indirect interdependencies is important in the analysis and control of multi-class servers. Direct interdependencies have predictable behavior that can be accurately captured by an analytical model. An ideal time-sharing system, e.g., [19, 20], is a good example where a thread will run for its entire scheduling quantum, Q, without blocking. Unfortunately, this is seldom the case for web servers, especially with the growing popularity of per-user customization. Therefore, when requests require many I/O operations, the bottleneck is shifted from the CPU to the memory or to the disk; it becomes much harder to predict the impact of one service class on the other ones.

In some cases, precise understanding of indirect interdependencies may not be necessary. This occurs when requests from different service classes have similar resource requirements. The load on all of the resources (including the bottleneck one) will, thus, be proportional to the number of requests that are being concurrently processed in each service class. We refer to these workloads as *homogeneous*. For example, a server that wants to provide client-side differentiation can be configured with several service classes, one for each group of clients. We, therefore, expect that these service classes will have similar service rates, μ_i, and speedup functions, $G_i(m)$, but with possibly different arrival rates, λ_i. Alternatively, when very different workloads need to be managed on the same system, e.g., a web server and an FTP server, each incoming request may have very different resource requirements. In this case, we refer to the workloads as *heterogeneous*.

We study the multi-class server in the context of our workload categorization. Our goal here is to quantify the impact of increasing the concurrency of one service class on the performance of the other running services. We use a similar setup in Section II, but now, we run two independent Apache services. Each service can be configured to receive requests for one of the three workloads: static, dynamic, and mixed. In particular, we test two configurations. The first configuration reflects the homogeneous workload, where incoming requests to both Apache services are for the mixed workload; the second configuration reflects the heterogeneous workload, where one service is designated as the static workload and the other service is designated as the dynamic workload. Finally, we measure the maximum throughput as a function of the number of threads that are allocated to each service class.

Figure 5 reflects the throughput gain as the thread allocation of the first service class is increased while the allocation of the second class is held constant. Each line represents a different allocation for the second service class; the "No sharing" line indicates that there is only a single class running on the server.

The homogeneous workload behaved as expected, where the throughput of a service class is proportional to its thread allocation. Specifically, we can express the service rate of any service class as a function of the number of the threads that are running:

$$\mu(m) \approx \frac{m}{\sum_{i=1}^{n} m_i} G(\sum_{i=1}^{n} m_i)\ \mu, \tag{12}$$

where m_i is the class-i threads that are running, or equivalently, the number of requests that are being concurrently processed by service class S_i.

The heterogeneous workload, on the other hand, did not exhibit the same behavior. Here we fixed the number of threads that are allocated to the server with the dynamic workload ($SRV_{dynamic}$ for short) and increased the thread allocation of the server with the static workload (SRV_{static} for short). Based on our measurements, we observed three unexpected phenomena:

P1. Even when $SRV_{dynamic}$ is assigned a single thread, its impact on the performance on SRV_{static} is significant. In fact, we observed an artificial ceiling that limited the maximum performance of the SRV_{static}.

P2. When the thread allocation of $SRV_{dynamic}$ is increased, but still below its saturation point ($\leq$ 16 threads), SRV_{static} incurs a small decrease in performance.

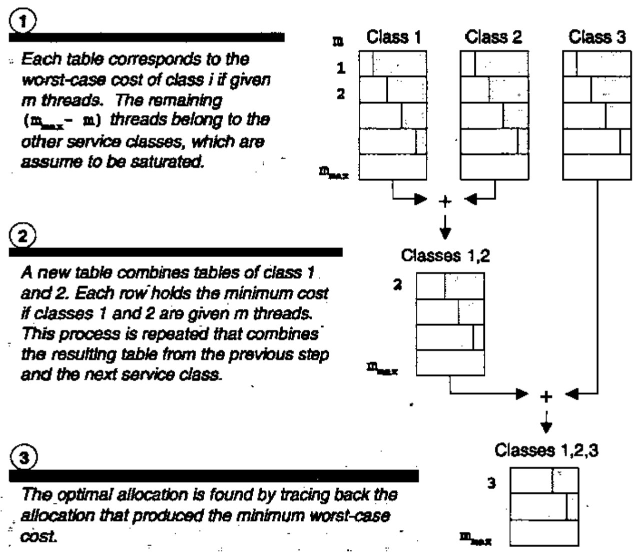
P3. After the thread allocation of $SRV_{dynamic}$ is increased beyond its saturation point, the SRV_{static} has a much greater performance drop. In both cases (P2 and P3), the performance drop is not proportional to the thread allocation of the two servers.

The above example shows an important result, namely, without precise understanding of the resource requirements of different and heterogeneous workloads, using thread allocation to provide QoS differentiation is not an effective approach. Fine-grain resource management must be used to provide effective QoS guarantees [6, 28, 32]. Unfortunately, these techniques require substantial changes to the application or the OS. We, however, show that if services are configured with homogeneous workload to provide client-side differentiation, then multi-threading can be used as an effective tool.

VI. PROVIDING QoS GUARANTEES

Providing QoS guarantees is motivated by the need to protect certain — possibly high-priority — service classes from others overloading the server. As shown in Section V, this is very difficult without explicit OS support, where strict resource limits are allocated to each service. In this section, we describe the extent to which thread allocation can be used to guarantee specific service delays. We will continue to focus on homogeneous workloads; our proposed technique is aimed at providing QoS guarantees to different client groups, each represented by a separate service class.

We focus in this section only on providing worst-case QoS guarantees. Here, the protected service is allocated the minimum number of threads such that no matter how high the load increases for the other service classes, it can still (statistically) meet its QoS objective. We studied a more general case in the extended version of this paper [16], where we also characterized the behavior of the system for multiple classes for any thread allocation. The derivation was based on extending the Markov chain that was presented in Section IV to be multi-dimensional, where an additional dimension was necessary for each service class. We have found that when the system is lightly-loaded, thread allocation of a given service class is minimally affected by the allocation of the other service classes.



1
Each table corresponds to the worst-case cost of class i if given m threads. The remaining (m_max - m) threads belong to the other service classes, which are assume to be saturated.
m
1
2
m_max
Class 1
Class 2
Class 3
2
A new table combines tables of class 1 and 2. Each row holds the minimum cost if classes 1 and 2 are given m threads. This process is repeated that combines the resulting table from the previous step and the next service class.
Classes 1,2
3
The optimal allocation is found by tracing back the allocation that produced the minimum worst-case cost.
Classes 1,2,3

Fig. 6. Dynamic programming algorithm for finding the allocation that minimizes the worst-case cost of the system. The illustration is limited to three service classes.

We start by considering the system that is heavily-loaded such that all classes are overloaded except for a single one, where the term "overloaded" implies that all threads are continuously busy. Let k be the non-overloaded service class and the other $n-1$ classes have $m_j \approx m_j^0$, for $j \neq k$. Here we also consider $Q \to 0$. Since only class k is not overloaded, the single class case in Section IV can be used, but with a different state-dependent transition rates μ_i.

Going back to Figure 4, we can define μ_i as follows:

$$\mu^{(i)} = \frac{\mu}{i+\gamma_k} G(i+\gamma_k). \tag{13}$$

where $\gamma_k = \sum_{j=0, j\neq k}^{n} m_j^0$. Here μ is the same for all service classes as the workload is assumed to be homogeneous. The ratio $\frac{\mu}{i+\gamma_k}$ reflects the fact that each thread has to share the system with $i+\gamma_k-1$ other threads. The steady-state probabilities can be similarly defined:

$$p_i = \begin{cases} \frac{\lambda_k (i+\gamma_k)}{i\mu G(i+\gamma_k)} p_{i-1} & \text{for } i=1,\ldots,m_k^0 \\ \frac{\lambda_k (m_k^0+\gamma_k)}{m_k^0 \mu G(\gamma_k+m_k^0)} p_{i-1} & \text{for } i=m_k^0+1,\ldots,K_k. \end{cases} \tag{14}$$

The remaining derivation is similar to the one in Section IV and is omitted. The computed $\bar{Y}_k = (\sum_{i=0}^{K_k} i \cdot p_i)/\lambda_k$, thus, reflect the worst-case response time when all but class k are overloaded.

The expression for the worst-case response time can now be used to determine the thread allocation that can meet the desired QoS objective. We express the QoS here using the notion of holding cost. Formally, let $h_k(t)$ be the cost of a request in service class k as a function of its response time t. Using $h_k(t)$ gives us flexibility in defining different QoS objectives. For example, it allows us to assign separate weights to different service classes, which can be used to provide different QoS levels to a multi-class QoS application.[4] The holding cost function can be arbitrary, however, with the restriction of being a monotonically non-decreasing function of t.[5]

We now extend the notion of cost to any thread allocation $\mathbf{M}$. We first define $c_k(\mathbf{M})$ as the worst-case cost for service class k in $\mathbf{M}$. It is computed by assuming that all service classes except for class k are overloaded as:

$$c_k(\mathbf{M}) = h_k(\bar{Y}_k),$$

where $\bar{Y}_k$ is just the worst-case response time as computed above. The sum of these costs $c(\mathbf{M}) = \sum_j c_j(\mathbf{M})$ is defined to be the cost for allocation $\mathbf{M}$. The allocation that minimizes the worst-case cost is thus

$$\mathbf{M}_{min} = \min_{\mathbf{M}}\{c(\mathbf{M})\}. \tag{15}$$

To efficiently compute $\mathbf{M}_{\mathbf{min}}$, we first observe that our definition of γ_k in Eq. (13) does not distinguish between different thread allocations to the overloaded service classes. This allow us to use dynamic programming to solve for $\mathbf{M}_{min}$, where in each step we group all overloaded classes together and then find the allocation that minimizes the cost of non-overloaded classes.

The basic algorithm is outlined (graphically) in Figure 6, where the algorithm is divided into n steps. In the first step, n tables are created, one for each service class. Each table contains the expected cost for any thread allocation to its corresponding service class, given that all other service classes are overloaded. We only need m^{max} entries to capture this expected cost. The next step combines the tables for classes 1 and 2 into a new table by finding the minimum cost for each allocation given all possible combinations from the tables for classes 1 and 2. Each additional step then combines the resulting table from the previous step with the table of an additional service class. At the end, the final table will contain the minimum cost and by tracing back the allocation that produced it, we can determine $\mathbf{M}_{min}$.

VII. EVALUATION

A realistic server environment is used to verify the correctness of our derivations with respect to our original assumptions and also demonstrate that the proposed scheme makes near-optimal allocation of threads using our proposed techniques. We used a similar experimental setup to that in Section I. However, we configured a second Apache server as shown in Figure 7 to act as a separate service class. Three parameters describe each service: the arrival rate, λ_i, the listen queue length, B_i, and the thread allocation, m_i^0. In the presented experiments, we set B_i to 128 requests for $i=1,2$.

[4]In this case, each QoS level is defined in terms of worst-case response time.

[5]This restriction avoids the situation where requests with long response times have lower cost than those with short response times.

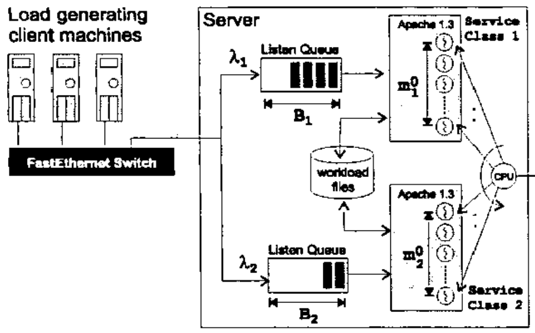


Fig. 7. Experimental Setup.

We have evaluated the system for different buffer lengths; in all cases, our results were consistent with those presented here.

Our evaluation is based primarily on response-time measurements at the client machines. This response time is the total wait time before a request is completed. It is the summation of three mostly independent components: connection-establishment latency, propagation delay, and processing delay. However, we are only interested in the effects of thread allocation on the processing-delay component. Thus, by keeping the first two components constant, we are able to obtain an unbiased view of the performance of the different thread allocations. We take two measures to minimize the variation in these two components. First, we made sure that the client-to-server network path is bottleneck-free. We also reduced the connection-establishment timeout such that any packet drop during that phase will not skew our results. We estimated that the error introduced by the first two components to be less than 2 msec. Finally, because we need to conduct a large number of experiments to cover the wide range of variable parameters, we limit each run to 5 minutes and each experiment was repeated 20 times.

Our evaluation is split into two experiments: the first validates the correctness of our derivation and the second measures the effectiveness of our optimal allocation policy. In all cases, we assume that workload is homogeneous, and hence, we only focus on the extent that the thread abstraction can be used to provide client-side QoS guarantees. The effects of heterogeneous workloads were studied in Section V. Finally, due to space limit, we only present the results for the mixed workload as it is considered a realistic representation of real server workloads.

A. Experiment 1: Model Validation

We compared our predicted values of the response time with the real measurements for the single-class server configuration. We used a single Apache service and varied the configuration parameters across two dimensions: arrival rate, λ, and thread allocation, m^0. This is shown in Figure 8, where each line represents the response time for a fixed allocation as the arrival rate is increased.

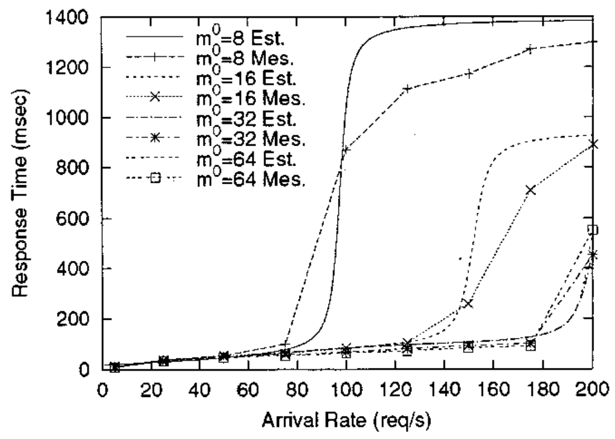


Fig. 8. Single class measurements

The figure shows that our derivations can accurately predict the expected performance of the underlying server for a wide range of configuration parameters. There is over-prediction for certain values of λ. This, we believe, is due to the system being critically-loaded. In particular, we can split the graph into two distinct regions: underloaded and overloaded. These are presented by the upper and lower parts of the S-shape of each line, respectively. The transition region between the underloaded and overloaded regions is very narrow and occurs when $\lambda \approx \mu G(m^0)$, where the system is critically-loaded.

Our analysis clearly exhibits the bimodal behavior of system queue occupancy. Namely, when the arrival rate is slightly below the saturation point, incoming requests are admitted almost immediately into service with little queueing delay. However, a slight increase in arrival rate can cause the delays to increase many folds simply because the system cannot keep up with incoming requests which causes queues to fill up. But since queues have limited capacity, the service delay is limited by the maximum length of such queues. The bimodal behavior raises an interesting design decision issue when configuring a web server, namely, when the system is underloaded, only a small queue is necessary to avoid request dropping. The length of the queue depends on the burstiness of arriving requests. However, once the system is overloaded, longer queues do not provide any performance advantage, but they increase the response time of accepted connections.

B. Experiment 2: QoS Guarantees

In Section VI, we described a dynamic programing algorithm to determine the thread allocation that can provide worst-case QoS guarantees. To verify our model, we must, unfortunately, test all possible thread allocations, which is computationally-prohibitive even with only two service classes. In this subsection, we look only at a single step of the algorithm, namely, given a thread allocation for a low-priority service class, we want to predict the thread allocation for a high-priority service class that can (statistically) guarantee a maximum response time. We will show that for each QoS requirement, our predictions are close to the measured values. With this, we can conclude the robustness of our algorithm in the general case.

Figure 9 shows the required number of threads for the high-priority service when its arrival rate is $\lambda_1 = 100$ reqs/s and the low-priority service is allocated a fixed number of threads. For instance, in the top plot where the low-priority service is allocated 8 threads, if a 1-second delay guarantee is required, then the high-priority service should be allocated at least 10 threads. The figure (for the measured and predicted lines) is computed by first assuming that the low-priority service is overloaded. A table that holds the thread allocation vs. worst-case response time for the high-priority service is then created. Finally, an inverse table lookup is used to determine the minimum allocation that meets the response time requirement.

The figure shows that our equation-based optimization is able to predict, with high accuracy, the thread allocation that achieves the minimum cost. One can see that if a similar process is used to create the initial tables in Section VI, then the resulting prediction will be close to the optimal value. We note that in this experiment we implicitly assumed a linear cost function where $h_k(t) = t$. Other cost functions can still be used.

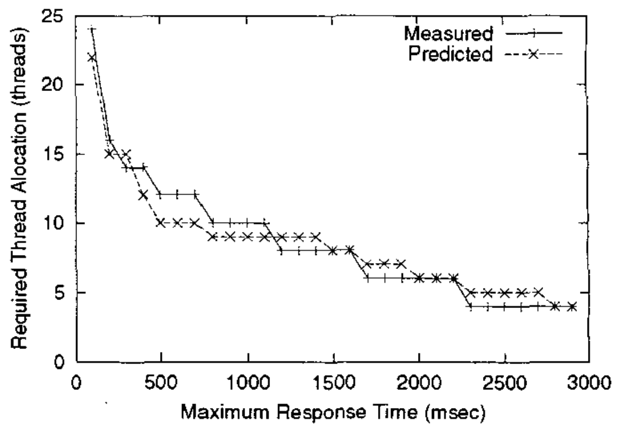
Overall, the above results show that our models are very robust. They capture the expected performance of a multi-threaded server as well as identify those instances where the model fails. Our approach can be used to improve the performance of existing QoS techniques.

VIII. RELATED WORK

The design and analysis of server QoS management techniques have been addressed extensively in the literature, for example, in [2, 4, 6, 10, 11, 13, 15, 22, 25, 32]. In general, our work complements existing QoS techniques by providing a rigorous analysis of one particular approach, namely, using the thread abstraction. Our focus is on determining the effectiveness and limitations of using thread-allocation to provide QoS guarantees or differentiation.

Several studies [1, 13, 22, 24, 25] have used thread allocation to provide application-level QoS. Vasiliou [30] focused his thread-based approach on providing a simple method for creating new scheduling disciplines. Similarly, Pandey [25] defined an object-oriented language to specify resource requirements for different client requests. Both [30] and [25] lack the translation between resource requirement and service quality. In this paper, we introduced the speedup function for this specific reason. We believe that it is a key element for determining the true performance for any thread-based allocation.

To handle changing load conditions, the authors of [1, 22–24, 32] proposed a feedback control mechanism to adjust the allocation of threads to different service classes based on on-line measurement of QoS metrics. Particularly in [24], a control-theoretic approach is used to implement a Proportional-Integrator (PI) controller that adapts to the load conditions on the server. Unfortunately, this type of analysis (linear control theory) is only suitable for steady request arrivals with predictable service demands. In this paper, we have shown the need for better monitoring techniques in the adaptation process. We have also shown when the dynamic adaptation will fail to provide the QoS differentiation.



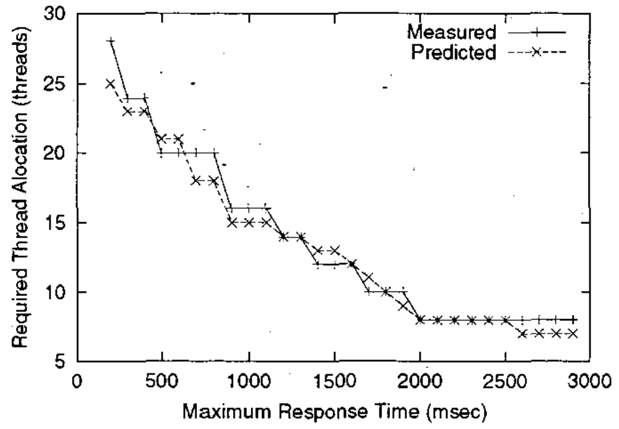


Fig. 9. Optimal allocation that guarantees response time guarantees for high-priority: (top) low-priority service is allocated 8 threads, and (bottom) low-priority service is allocated 16 threads.

IX. CONCLUSIONS

In this paper, we have provided a rigorous analysis of the performance of thread-based QoS support. We also presented an efficient optimization algorithm for determining the thread allocation, if any, that minimizes the system's cost, based on our economic formulation. Through empirical validation in real server environments, we showed that the derived results are applicable to real-world systems.

The results presented in this paper are essential to the design of any efficient thread-based QoS differentiation mechanism. Three important conclusions can be drawn from our study. First, based on the shape of the speedup function, we argue that dynamic adaptation of thread allocation based on the response-time measurements only is not sufficient to guarantee the stability of the control mechanism. The controller must continuously monitor the saturation point, which may shift with changing workloads. Second, indirect interdependencies between threads that arise from non-trivial sharing of system's resources can yield unpredictable performance interactions. We have shown that even with a small number of threads dedicated to I/O-heavy workloads, the performance of other running service can be affected significantly. Therefore, without accurate understanding of resource requirements, the thread abstraction alone cannot provide the necessary QoS guarantees, or even QoS differentiation, to running services. Finally, when similar or homogeneous services are being hosted on a single server to provide client-side differentiation, the thread abstraction can be used to provide effective and predictable statistical QoS guarantees.

REFERENCES

[1] T. Abdelzaher and N. Bhatti, "Web Content Adaptation to Improve Server Overload Behavior," in *International World Wide Web Conference*, May 1999.

[2] J. Aman, C. K. Eilert, D. Emmes, P. Yocom, and D. Dillenberger, "Adaptive Algorithms for Managing Distributed Data Processing Workload," *IBM Systems Journal*, vol. 36, no. 2, pp. 242–283, 1997.

[3] M. F. Arlitt and C. L. Williamson, "Web server workload characterization: The search for invariants," in *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, April 1996.

[4] C. Aurrecoechea, A. Campbell, and L. Hauw, "A Survey of QoS Architectures," *Multimedia Systems*, vol. 6, no. 3, pp. 138–151, 1998.

[5] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. Katz, "TCP Behavior of a Busy Internet Server: Analysis and Improvements," in *Proc. of IEEE INFOCOM '98*, March 1998, pp. 252–262.

[6] G. Banga, P. Druschel, and J. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Third Symposium on Operating Systems Design and Implemenation*, February 1999, pp. 45–58.

[7] P. Barford, A. Bestavros, A. Bradley, and M. E. Crovella, "Changes in Web Client Access Patters: Characteristics and Caching Implications," in *World Wide Web, Special Issue on Characterization and Perfromance Evaluation*, 1999, pp. 15–28.

[8] P. Barford and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," in *Proceedings of Performance'98/ACM Sigmetrics'98*, May 1998, pp. 151–160.

[9] D. Bertsekas and R. Gallager, *Data Networks*. Prentice Hall, 1992.

[10] N. Bhatti and R. Friedrich, "Web Server Support for Tiered Services," *IEEE Network*, vol. 13, no. 5, pp. 6764–71, Sep.–Oct. 1999.

[11] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Retrofitting Quality of Service into a Time-Sharing Operating System," in *USENIX Annual Technical Conference*, June 1999, pp. 15–26.

[12] S. D. Committee, "SPECweb," Tech. Rep., April 1999, http://www.specbench.org/osg/web/.

[13] L. Eggert and J. S. Heidemann, "Application-Level Differentiated Services for Web Servers," *World Wide Web*, vol. 2, no. 3, pp. 133–142, 1999.

[14] A. Feldmann, *Characteristics of TCP Connection Arrivals*, ser. Self-Similar Network Traffic and Performance Evaluation. John Wiley and Sons, Inc., 2000, ch. 15, pp. 367–399.

[15] Hewlett Packard Corp., "WebQoS Technical White Paper," 2000. http://www.internetsolutions.enterprise.hp.com/ webqos/products/overview/wp.html.

[16] H. Jamjoom, C.-T. Chou, and K. G. Shin, "The Impact of Concurrency Gains on the Analysis and Control of Multi-threaded Internet Services," University of Michigan Technical Report, Tech. Rep. CSE-TR-480-03, 2003.

[17] H. Jamjoom and K. G. Shin, "Eve: A Scalable Network Client Emulator," University of Michigan Technical Report, Tech. Rep. CSE-TR-478-03, 2003.

[18] S. Keshav, *An Engineering Approach to Computer Networking*. Addison-Wesley Publishing Company, 1997.

[19] L. Kleinrock, "Time-Shared Systems: A Theoretical Treatment," *Journal of the ACM*, vol. 14, April 1967.

[20] L. Klienrock, *Queueing Systems, Volume II: Computer Applications*. Wiley Interscience, 1976.

[21] C. M. Krishna and K. G. Shin, *Real-Time Systems*. McGraw-Hill, 1997.

[22] K. Lakshman, R. Yavatkar, and R. Finkel, "Integrated CPU and Network I/O QoS Management in an Endsystem," in *Proc. 5th International Workshop on Quality of Service (IWQOS'97)*, 1997, pp. 167–178.

[23] X. Liu, L. Sha, Y. Diao, J. L. Hellerstein, and S. Parekh, "Online Response Time Optimization of Apache Web Server," in *Proc. 11th International Workshop on Quality of Service (IWQOS 2003)*, 2003, pp. 461–478.

[24] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son, "A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers," in *IEEE Real-Time Systems Symposium*, Taipei, Taiwan, December 2001.

[25] R. Pandey, J. F. Barnes, and R. Ollsson, "Supporting Quality of Service in HTTP Servers," in *Symposium on Principles of Distributed Computing*, 1998, pp. 247–256.

[26] J. Reumann, H. Jamjoom, and K. G. Shin, "Adaptive Packet Filters," in *Proceeding of IEEE GLOBECOM'01*. San Antonio, Texas: IEEE, November 2001.

[27] S. Sarvotham, R. Riedi, and R. Baraniuk, "Connection-level Analysis and Modeling of Network Traffic," in *Proceedings of the ACM SIGCOMM Internet Measurment Workshop*, November 2001.

[28] O. Spatscheck and L. L. Peterson, "Defending Against Denial of Service Attacks in Scout," in *Third Symposium on Operating Systems Design and Implementation*, February 1999, pp. 59–72.

[29] H. Takagi, *Analysis of Polling Systems*. MIT Press, 1986.

[30] N. Vasiliou and Hanan, "Providing a Dierentiated Quality of Service in a World Wide Web Server," to Appear in Performance Evaluation Review.

[31] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra, "Kernel Mechanisms for Service Differentiation in Overloaded Web Servers," in *Proceedings of the 2001 Annual Technical Conference*, June 2001, pp. 189–202.

[32] M. Welsh, D. Culler, and E. Brewer, "SEDA: An Architecture for Well-Conditioned Scalable Internet Service," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[33] R. W. Wolff, *Stochastic Modeling and the Theory of Queues*. Prentice-Hall, Inc., 1989.