Dynamic Thread Assignment in Web Server Performance Optimization

Wemke van der Weij, Sandjai Bhulai, and Rob van der Mei

CWI
Kruislaan 413
P.O. Box 94079
1090 GB Amsterdam
The Netherlands
{weij, mei}@cwi.nl

Vrije Universiteit Amsterdam
Faculty of Sciences
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
sbhulai@few.vu.nl

Abstract

Popular web sites are expected to handle huge amounts of requests concurrently within a reasonable timeframe. The performance of these web sites is largely dependent on effective thread management of their web servers. Although the implementation of static and dynamic thread policies is common practice, remarkably little is known about the implications on the performance. Moreover, the commonly used policies do not take into account the complex interaction between the threads that compete for access to a shared processor resource.

We propose new dynamic thread-assignment policies that minimize the average response time of web servers. The web server is modeled as a two-layered tandem of multi-threading queues, where the active threads compete for access to a common hardware resource. Our results show that the optimal dynamic thread-assignment policies yield strong reductions in the response times. Validation on an Apache web server shows that our dynamic thread policies confirm our analytical results.

1 Introduction

The rise of Internet and broadband communication technology have boosted the use of web-based services that combine and integrate information from geographically distributed information systems. As a consequence, popular web sites are expected to handle huge numbers of requests simultaneously without noticeable degradation of the response-time performance. Moreover, web servers must perform significant CPU-intensive processing, caused by the emergence of server-side scripting technologies (e.g., Java servlets, Active Server Pages, PHP). Furthermore, web pages involving recent and personalized information (location information, headline news, hotel reservations) are created dynamically on-the-fly and hence are not cacheable. This limits the effectiveness of caching infrastructures that are usually implemented to boost the response-time performance of commercial web sites and to limit bandwidth consumption. At the same time, as a result of the recent advances in wired networking technology, there is usually ample core network bandwidth available at reasonable prices. As a consequence of these developments, web servers tend to become performance bottlenecks in many cases; examples of badly performing web

sites appear in the newspapers on a regular basis. These observations raise the need for web-based service providers to control the performance of their web servers.

Web servers are typically equipped with a pool of threads. In many cases, a request is composed of a number of processing steps that are performed in sequential order. For example, an HTTP GET request may require processing in several steps: a document-retrieval step and a sequence of script-processing steps to create dynamic content. Similarly, an HTTP POST request may require a document-processing step and several database update queries. To handle the incoming requests, web servers usually implement a number of thread pools that are dedicated to process a specific processing step [1].

The performance of the web server is largely dependent on the thread-management policy. This policy may be either static (i.e., with a fixed number of threads – possibly of different types) or dynamic (i.e., where threads may be created or killed depending on the state of the server). Traditionally, many web servers implement a simple static thread-assignment policy, where the size of the thread pool (i.e., the maximum number of threads that can simultaneously execute processing steps) is a configurable system parameter. This leads to a trade-off regarding the proper dimensioning of thread pools to optimize performance: on the one hand, assigning too few threads may lead to relative starvation of processing power, creating a performance bottleneck that may increase the average response time of requests, particularly when the workload increases. On the other hand, if the total number of threads running on a single hardware component is too large, performance degradation may occur due to superfluous context switching overhead and memory or disk I/O activity. Nowadays, more efficient thread policies are widely implemented. In order to effectively react to sudden bursts of transaction requests, many web servers implement simple dynamic thread-management algorithms that allow threads to be created or killed, depending on the actual number of active threads. However, even though the implementation of these thread policies is common practice, a thorough understanding of the implications of the proper choice of thread-assignment policies and the settings of the parameters on the performance of the web server is mostly lacking. In particular, the trade-off between relative starvation of processing power in the case of too few threads and the performance degradation in the case of too many threads is not well understood. Moreover, the commonly used thread policies do not take into account the probability distribution of the service times required by the different requests, while significant performance improvements can be obtained by doing so.

A key feature of multi-threaded web servers is that the threads typically share a common processor (CPU) with a limited amount of processor capacity. This naturally leads to the formulation of a *two-layered tandem of multi-server queues*, where the active threads share the processor capacity in a processing sharing fashion; i.e., when there are k threads active at some moment in time, then each of these k threads has a fair share $1/k$ of the total processor capacity [1]. In this model, transaction requests are represented by customers, threads are represented by servers, and response times are represented by the sojourn times of the customers. To identify optimal thread-assignment policies, we describe the evolution of the system as a Markov decision model and derive optimal thread policies from the properties of the relative value function. In doing so, we show that the structure of the optimal thread policy strongly depends on the service-time distributions of the different processing steps in the web server; in practice, these distributions can be monitored and

updated on-the-fly.

An interesting feature of this model is that it has a two-layered structure, modeling the complex *interaction* between contention at the *hardware* (CPU) and the *software entities* (threads) layer. At the software layer, the processing steps, comprising a request, are processed by different, say N, types of threads. However, each of the active threads effectively shares the underlying processor capacity: the more threads are active, the smaller the processor capacity is assigned to each thread. In this way, the thread is no longer an autonomous entity operating at a fixed rate; instead, the processing rate of each thread continuously changes over time. Evidently, for $N = 1$, the model coincides with the classical processor-sharing discipline; but for $N > 1$, the processing speed of one thread pool depends on the state of the other thread pools. This type of interaction makes the model rather complicated.

1.1 Related Literature

Although the theory of job scheduling with autonomous independent servers is well-matured, in the literature only a few papers deal with scheduling of web servers. Harchol-Balter et al. [2, 3] and Crovella et al. [4] study scheduling policies for web servers to reduce the response-time performance of web servers with static web pages, provided the size of a web page is known *a-priori*; for this type of models, the results show that the classical Shortest Remaining Processing Time (SRPT) policy is very effective [5]. In contrast to the present paper, it should be noted that the results in [2, 3] are based on the assumption that the network interface, rather than the web server itself, is the performance bottleneck; this leads to fundamentally different performance models than the one considered in the present paper.

Detailed performance models for web servers, explicitly including the interaction between software and hardware contention, were proposed in [1, 6]. A limited number of papers focus on these queueing networks with a layered structure. Rolia and Sevcik [7] propose the Method of Layers (MoL), i.e., a closed queueing-network model based on the responsiveness of client-server applications. Woodside et al. [8] propose the so-called Stochastic Rendez-Vous Network (SRVN) model to analyze the performance of application software with client-server synchronization. Related models are the so called coupled-processor models, i.e., multi-server models where the speed of a server at a queue depends on the number of servers at the other queues (see [9, 10, 11]). For a two-layered network of two multi-server queues with processor sharing, remarkable results on the per-queue stability were obtained in [12]. To the best of the authors' knowledge, the problem of dynamic thread assignment in layered queueing networks has not been addressed in the literature.

1.2 Contribution

In this paper we model a web server by a two-layered queueing network with a shared processor resource. We describe the evolution of the system as a Markov decision process from which we obtain simple and readily implementable dynamic thread-assignment policies that minimize the expected response time of the requests. The service-time distributions are modeled by the class of phase-type distributions, which is a broad class of

distributions and also allows to study the impact of heavy-tailed distributions. The results show not only *that*, but also *how* the optimal policy depends on the service-time distributions at each of the processing steps. The proposed policy uses monitored information on both the number of active threads and the probability distribution of the required service time per request. Our results show that the optimal dynamic thread-assignment policies yield strong reductions in the response times. To validate the model, we have tested the performance of our policies in an experimental setting on an Apache web server. The experimental results show that our policies indeed lead to significant reductions of the response time, which demonstrates the practical usefulness of the results.

1.3 Outline

The remainder of this paper is organized as follows. In Section 2 we formulate the model. Section 3 derives optimal dynamic thread-assignment policies. In Section 4 we consider numerical experiments and evaluate them on an Apache web server. We conclude in Section 5 and give ideas for further research directions.

2 Model description

In this section we model the problem of dynamic thread assignment in the context of a multi-layered queueing system with a shared resource. For this purpose, consider a network of N queues in tandem with a single shared processor for serving arriving requests. Requests arrive according to a Poisson process with rate λ to the first queue. At each queue, threads can be spawned which may be assigned to a request. When a request is assigned to a thread at queue i, it receives service S_i with mean duration β_i for $i = 1, \ldots, N$. However, during service, the request only gets a fraction of the total capacity of the server, depending on the number of outstanding threads $k^{(i)}$ at each queue i for $i = 1, \ldots, N$. Upon completion of service, the thread is terminated and the request proceeds to queue $i+1$ if $i < N$, or it leaves the system otherwise. If a request is not assigned a thread, the request joins an infinite buffer at the queue and waits until it is assigned a thread. Note that we do not explicitly model delays due to context switching between threads, since the CPU time in comparison to the processing times of the threads is neglible (see Remark 4.1 for a justification).

To obtain optimal thread-assignment policies that minimize the expected response times, we model the system in the framework of Markov decision theory. To this end, we model the service-time distribution of S_i by a phase-type distribution. Consequently, consider a Markov chain with $M_i + 1$ states having probability distribution $\eta^{(i)}$ defined on states 1 to M_i for the initial state. When the Markov chain is in state j, the time that the process spends in state j has an exponential distribution with parameter $\mu_j^{(i)}$. After this exponentially distributed time, the process jumps to state l with probability $p_{jl}^{(i)}$, or jumps to the absorbing state $M_i + 1$ with probability $p_{j,M_i+1}^{(i)}$. The absorbing state corresponds to a completion of the service period at that queue.

Phase-type distributions have the important feature that they are dense in the class of all non-negative distributions, while retaining their tractability [13]. Therefore, it is possible to model heavy-tailed distributions by phase-type distributions. This is especially

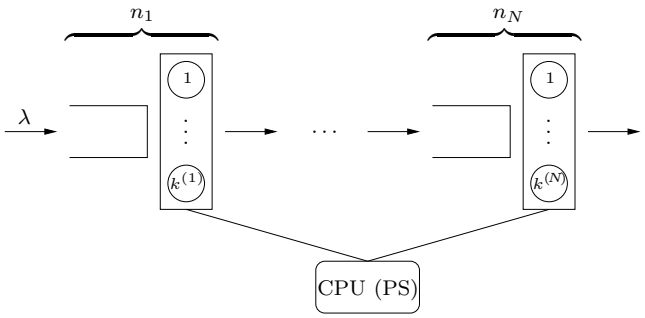

Figure 1: A two-layered queueing system

relevant, since it has been observed that file sizes on web servers follow a heavy-tailed distribution (see, e.g., [4]). It is common to fit phase-type distributions on the mean $\mathbb{E}S_i = \beta_i$ and on the coefficient of variation c_{S_i} (see, e.g., [14]), or by the more complex EM-algorithm (see [15]).

Next, we uniformize the system (see Section 11.5 of [16]). For simplicity we assume that $\lambda + \max\{1/\beta_1, \ldots, 1/\beta_N\} = 1$; we can always get this by scaling. Uniformizing is equivalent to adding dummy transitions (from a state to itself) such that the rate out of each state is equal to 1; then we can consider the rates to be transition probabilities. Note that rate costs in this case are equivalent to lump costs at each epoch.

Let $\vec{n}$ be the vector that denotes the number of requests in each phase, i.e., $n_j^{(i)}$ is the number of requests in phase j that are waiting at queue i plus the number of requests in phase j in service at queue i for $i = 1, \ldots, N$. Moreover, let the vector $\vec{k}$ denote the number of outstanding threads, i.e, $k_j^{(i)}$ is the number of outstanding threads for requests in phase j at queue i. Thus, the number of outstanding threads at queue i equals $k^{(i)} = k_1^{(i)} + \cdots + k_{M_i}^{(i)}$. A state x of the system, depicted in Figure 1, is then given by the tuple $(\vec{n}, \vec{k})$.

The goal is to minimize the expected number of requests in the system, which directly relates to minimizing the expected response time using Little's Law (see [14]). Therefore, the system is subject to unit costs for holding a request per unit of time in the system. Let $u_t(x)$ denote the total expected costs up to time t when the system starts in state x. Note that the Markov chain satisfies the unichain condition, so that the average expected cost $g = \lim_{t\to\infty} u_t(x)/t$ is independent of the initial state x (Proposition 8.2.1 of [16]).

We define the dynamic programming operator T as follows:

$$TV(\vec{n}, \vec{k}) = \sum_{i=1}^{N} \sum_{j=1}^{M_i} n_j^{(i)} + \lambda \sum_{j=1}^{M_1} \eta_j^{(1)} H\big(\vec{n} + e_j^{(1)}, \vec{k}\big) + \sum_{i=1}^{N} \sum_{j=1}^{M_i} \sum_{l=1}^{M_i} \frac{k_j^{(i)} \mu_j^{(i)} p_{jl}^{(i)} \, H\big(\vec{n} - e_j^{(i)} + e_l^{(i)}, \vec{k} - e_j^{(i)} + e_l^{(i)}\big)}{k^{(1)} + \cdots + k^{(N)}} +$$

$$\sum_{i=1}^{N}\sum_{j=1}^{M_i}\sum_{l=1}^{M_i}\frac{k_j^{(i)}\mu_j^{(i)}p_{j,M_i+1}^{(i)}\eta_l^{(i+1)}}{k^{(1)}+\cdots+k^{(N)}}H\big(\vec{n}-e_j^{(i)}+e_l^{(i+1)},\vec{k}-e_j^{(i)}\big)+$$

$$\left[1-\lambda-\sum_{i=1}^{N}\sum_{j=1}^{M_i}\sum_{l=1}^{M_i+1}\frac{k_j^{(i)}\mu_j^{(i)}p_{jl}^{(i)}}{k^{(1)}+\cdots+k^{(N)}}\right]V(\vec{n},\vec{k}),$$

with e_i the unit vector with all entries zero, except for the i-th entry for $i=1,\ldots,N$, and with e_{N+1} the zero vector. The unit vector $e_j^{(i)}$ is similarly defined. The actions, denoted by H, are given by

$$H(\vec{n},\vec{k})=\min\left\{V\big(\vec{n},\vec{k}+a\,e_j^{(i)}\big)\;\middle|\;\begin{array}{l}i=1,\ldots,N,\\ j=1,\ldots,M_i,\end{array}\quad a\in\mathbb{N}_0\right\},$$

with $\mathbb{N}_0=\{0,1,2,\ldots\}$.

The first term in the expression $TV(\vec{n},\vec{k})$ models the direct costs, i.e., the total number of requests in the system. The second term models the arrivals, which occur with rate $\lambda\eta_j^{(1)}$ to phase j at the first queue. The transitions of a request to a different phase within each queue are given by the third term. A transition from phase j to phase l for a request in queue i occurs with rate $\mu_j^{(i)}p_{jl}^{(i)}$, but this is adjusted with the factor $k_j^{(i)}/\big(k^{(1)}+\cdots+k^{(N)}\big)$, since that request only uses a fraction of the service capacity. The next term, which accounts for a transition to the absorbing state, is similarly explained with the exception that the departure is split into arrivals to phase l of the next queue with probability $\eta_l^{(i+1)}$. The last term is the uniformization constant to account for the dummy transitions added to the model.

Note that the system is specifically modeled such that when a request moves from one phase to another, the previously assigned thread is not lost. The thread is only released upon completion of the service period. From the definition of H, we see that it is possible to spawn more threads than there are requests waiting. This is obviously not optimal, since that will lead to loss of capacity in the model. Therefore, the model ensures that threads will only be spawned for requests that are waiting for service. Also observe that upon arrival of a request, the system directly knows in which service phase the request will start. This is not unrealistic to assume, since web servers can induce this information from the HTTP request headers.

In the above set of equations, the function V is called the relative value function. The relative value function $V(\vec{n},\vec{k})$ has the interpretation of the asymptotic difference in total costs that results from starting the process in state $(\vec{n},\vec{k})$ instead of some reference state. The long-term average optimal actions are a solution of the optimality equation (in vector notation) $g+V=TV$. Another way of obtaining them is through value iteration, by recursively defining $V_{l+1}=TV_l$ for arbitrary V_0. For $l\to\infty$, the maximizing actions converge to the optimal ones (for existence and convergence of solutions and optimal policies we refer to [16]). Consequently, when V is known, we can restrict our attention to the function H to obtain the optimal actions.

3 Dynamic thread management

In this section we focus on dynamic thread management. We determine, using dynamic programming, optimal policies minimizing the expected response time per request. The performance of the optimal policies is compared to the performance of policies that only serve requests based on the number of threads outstanding. A specific example of the latter case is the policy that serves one request with only one outstanding thread until it leaves the system, resulting in a first-come-first-served (FCFS) policy. The other extreme is the policy that always serves all requests so that new threads are spawned for arriving requests. The intermediate case, which is commonly implemented in web servers, is the policy that serves requests simultaneously with a number of threads of which the maximum number is limited by some specified number. More precisely, let $\pi^{(k)}$ be the policy that spawns at most k threads in total such that the requests in queue i get priority over requests in queue j when $i \geq j$. Note that the three policies mentioned earlier are represented by $\pi^{(1)}$, $\pi^{(\infty)}$, and $\pi^{(k)}$ for $k = 2, 3, \ldots$, respectively.

Exponential service-time distributions are a special case of phase-type distributions, namely those with one phase only. They accurately model the service times of simple HTTP requests, e.g., a GET request for an HTML document. In this case, the optimal policy that minimizes the expected response time is to serve according to policy $\pi^{(1)}$, i.e., serve one request with only one outstanding thread until it leaves the system, such that the requests in queue i get priority over requests in queue j when $i \geq j$. This result also holds when the service-time distributions are replaced with Erlang distributions. This class of distributions models the situation where a web server fetches a web page and also performs server-side scripting for the page.

The optimal policy changes when the service-time distributions are replaced with hyper-exponential distributions. In this case, requests waiting in queue i may overtake requests in queue $j \geq i$ when a thread is opened at station i. It turns out that assigning a new thread is only optimal when overtaking a request further in the queueing network is possible. Therefore, the optimal policy can be obtained efficiently by recursive computation. Note that this policy coincides with the optimal policies for exponential and Erlang service-time distributions. For these distributions there is only one service phase, so that requests cannot overtake each other. Therefore, there is only one thread outstanding at most.

In the previous paragraphs, we have obtained intuition for optimal policies for dynamic thread management. We have seen that overtaking of requests plays a key role in the decision to spawn threads. In the case of exponential and Erlang service distributions it was not possible to overtake when spawning additional threads, and therefore the FCFS policy is optimal. In the case of hyper-exponential service distributions, we obtained that the optimal actions at queue i depend on the state of queues $j \geq i$ and the decision rules for those queues. This result also holds for the general phase-type service distributions.

Theorem 3.1: Let $\varphi_i(\vec{n}, \vec{k})$ denote the decision rule that describes the thread-management rule at queue i, for $i = 1, \ldots, N$. Let φ_i be such that it spawns s threads for requests in phase j at queue i given state $(\vec{n}, \vec{k})$, when the s phase-j requests overtake in expectation requests in queue $j \geq i$ under decision rules $\varphi_{i+1}, \ldots, \varphi_N$. Then policy $\pi = (\varphi_1, \ldots, \varphi_N)$ is optimal.

4 Numerical experiments

In the previous section, we determined optimal policies for general phase-type service distributions. In this section, we compare these policies with other thread-assignment rules that are frequently used. First, for various parameter settings, we analytically show that the optimal policies outperform the simple thread-assignment rules. Then, we compare the theoretically obtained improvements with those that are obtained in an experimental setting on an Apache web server.

4.1 Comparison of policies

In this subsection, we analytically compare the optimal policy, which we denote by π^*, with commonly used alternative policies. For this purpose, we use a web server infrastructure with $N = 2$, so that overtaking can occur. Moreover, for this infrastructure the state space is still of reasonable size so that the computation of the expected response times is numerically tractable. We consider the following alternative policies: $\pi^{(1)}$, $\pi^{(4)}$, $\pi^{(\infty)}$, and $\pi^{(1,1)}$. Note that the first three policies follow the notation given in Section 3, i.e., the policy that spawns at most $k = 1, 4$, and unlimited threads in total such that requests in queue i get priority over requests in queue j when $i \geq j$. The last policy, denoted by $\pi^{(1,1)}$, is the policy that spawns at most one thread at queue 1 and at most one thread at queue 2, independent of each other. We are interested in the gain $\big(\mathbb{E}W(\pi^{(s)}) - \mathbb{E}(\pi^*)\big)/\mathbb{E}W(\pi^*)$, where $\mathbb{E}W(\pi)$ is the expected response time under policy π, and $\pi^{(s)}$ is one of the alternative policies.

In our scenarios, we focus on exponentially and hyper-exponentially distributed service times. The choice for these distributions is motivated by the fact that they have a coefficient of variation that is equal to one and bigger than one, respectively. The coefficient of variation can be seen as a measure for the variation in the service times, i.e., it is the ratio of the standard deviation and the mean of the service times. Low (high) values of the coefficient of variation correspond to low (high) variability in the service times. These service distributions are rich and simple enough to gain insight into the structure of optimal policies. The Erlang distributed service times (which have a coefficient of variation smaller than one) are not considered here, because the optimal policy for the case of Erlang and exponentially distributed service times are equal.

In Table 1 we present the different scenarios with the corresponding parameter settings for which we have compared the policies. In case the coefficient of variation equals one, we only mention $\beta_1^{(i)}$, since there are no other phases. Moreover, the average load $\rho = \lambda(\beta_1 + \beta_2)$ on the system is taken to be constant, $\rho = 0.6$. The table also presents the gains in expected response times for the 12 different cases. The last line in this table represents the average gain compared to each policy.

Figure 2 shows the expected response times for the five policies. We can immediately see that the optimal policy π^* leads to significant reductions in the expected response time. For exponentially distributed service times at both queues, the optimal policy is given by $\pi^{(1)}$, and this can be seen in the figure by the two bars of equal height. However, we see that in many cases of hyper-exponentially distributed service times, the gain is significant compared to all other policies.

case	$c^2_{S_1}$	$c^2_{S_2}$	$r_1^{(1)}$	$r_1^{(2)}$	$\beta_1^{(1)}$	$\beta_2^{(1)}$	$\beta_1^{(2)}$	$\beta_2^{(2)}$	$\pi^{(1)}$	$\pi^{(4)}$	$\pi^{(\infty)}$	$\pi^{(1,1)}$
1	1	1			1.00		1.00		0.00%	13.82%	17.63%	16.92%
2	1	5		0.91	1.00		0.55	5.45	18.99%	5.73%	3.45%	47.38%
3	5	1	0.91		0.55	5.45	1.00		34.79%	19.77%	17.19%	53.05%
4	5	5	0.91	0.91	0.55	5.45	0.55	5.45	66.06%	23.78%	14.50%	94.96%
5	1	1			1.00		5.00		0.00%	7.12%	9.08%	10.02%
6	1	5		0.91	1.00		2.75	27.25	78.61%	14.73%	2.04%	97.35%
7	5	1	0.91		0.55	5.45	5.00		5.57%	9.93%	11.11%	14.74%
8	5	5	0.91	0.91	0.55	5.45	2.75	27.25	99.82%	26.93%	11.94%	119.78%
9	1	1			5.00		1.00		0.00%	7.12%	9.%	4.57%
10	1	5		0.91	5.00		0.55	5.45	0.00%	4.13%	5.25%	5.74%
11	5	1	0.91		2.75	27.25	1.00		98.91%	40.53%	13.64%	104.17%
12	5	5	0.91	0.91	2.75	27.25	0.55	5.45	91.26%	21.49%	7.14%	98.07%
									41.17%	16.25%	10.17%	55.56%

Table 1: The performance of policies under twelve scenarios.

4.2 The Apache web server

In this subsection we validate the theoretically obtained improvements of the previous section with improvements that are obtained in an experimental setting on an Apache web server. For the experimental setup, we use the Apache HTTP server version 1.3.33 running on a 2.8 GHz Linux platform with kernel version 2.4.31. The requests are generated according to a Poisson process by a Perl script that issues HTTP GET requests from a remote desktop. The requests that the script makes are requests to PHP pages that draw a random number w from a pre-specified probability distribution. This random number is then used to generate a file of size w megabytes, and is displayed as a web page. After displaying the web page, a second PHP page is requested which behaves similarly. The second page represents the requests at the second queue that also use the same underlying CPU, memory, and I/O hardware.

The policies are not implemented directly in the Apache web server code. The script that issues the requests for the web pages keeps track of the requests in service and does request policing. Thus, it maintains a list of requests that still need to be issued and implements a queue. Therefore, the script has complete state information and can decide when to issue a request for a web page with the right parameters, based on the given policies. Since the time the script needs for decision making is neglible, we expect that implementing the code in the Apache web server does not add significant additional computational overhead. Therefore, the results still give realistic indications of the improvements that can be obtained.

Remark 4.1: Note that the threads are spawned by the Apache web server itself, and consequently delays due to context switching between threads are taken into account in these experiments. In practice, when the number of threads increases, other hardware resources may become a bottleneck (e.g., memory or disk I/O). However, the optimal policies spawn only a finite number of threads such that these phenomena do not occur in our experiments. This leads to having increased queue sizes while the web server remains stable. Hence, we advise to use admission control based on the queue sizes such that the queues remain stable.

Presently, the Apache web server consists of a Multi-Processing Module (MPM) that implements a hybrid multi-process multi-threaded server. This module uses threads to serve requests with less system resources than a process-based server. The maximum total number of threads that may be spawned is equal to the parameter `MaxClients`, which is set to 150 in the standard configuration. The configuration file of the Apache web server

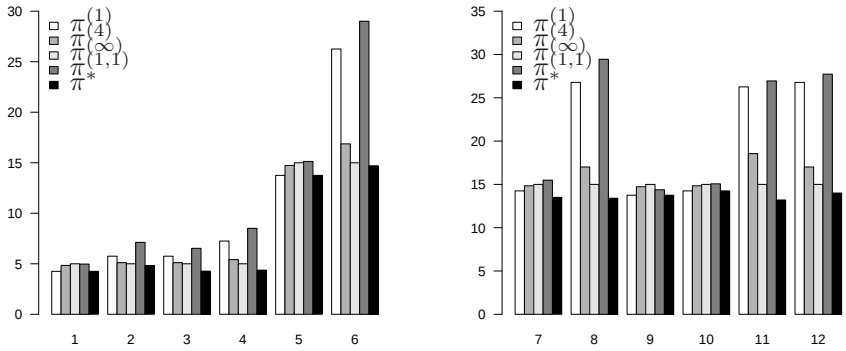

Figure 2: Expected response times

advises to set this number high so that maximum performance can be achieved (thus, effectively implementing $\pi^{(\infty)}$). In addition, Apache always tries to maintain a pool of spare or idle server threads, which stands ready to serve arriving requests. In this way, requests do not need to wait for new threads to be created before they can be served. Consequently, the assumption in our model that creating (killing) threads does not cost additional time is justified.

In Table 2 the gains of using policy π^* over policy $\pi^{(1)}$ and $\pi^{(\infty)}$ are listed. We compare π^* only with these two policies, since the results in Table 1 suggest that the best alternative policy is achieved either under $\pi^{(1)}$ or $\pi^{(\infty)}$. As mentioned in the previous paragraph, the policy $\pi^{(\infty)}$ also coincides with the standard thread-management policy used by the Apache web server. In Figure 3 the gains of the different cases are compared to the theoretically calculated gains. The figure shows that the observed gains closely match the theoretical gains, so that the multi-layered queueing model can be used to establish effective thread management policies in practice.

5 Conclusions and further research

We have considered the important problem of dynamic thread assignment in web servers such that the expected response time is minimized. This problem, in the context of multi-layer queueing networks, has received little attention. Consequently, the performance of thread-assignment policies is not well understood. We show that for phase-type service-time distributions the optimal policy spawns a thread for a request if that request can overtake a request in the queueing system. This insight, when using dynamic policies that have information on the service-time distributions, leads to an efficient recursive computation of the optimal policy. When the performance of this optimal policy is compared to the performance of policies that serve requests only based on the number of outstanding threads, it is shown that significant gains can be obtained. Experiments on an Apache web server show that the theoretically predicted gains are also achieved in practice.

We mention a number of interesting avenues for further research. First, for many transaction-based applications, the user-perceived performance is not fully described by the expected response time. The variability, and in many cases, even the tail probabilities of

case	$\pi^{(1)}$	$\pi^{(\infty)}$
1	0.00%	16.01%
2	17.55%	3.41%
3	31.62%	16.03%
4	60.23%	10.42%
5	0.00%	8.78%
6	75.56%	1.13%
7	5.12%	10.91%
8	92.76%	6.47%
9	0.00%	7.98%
10	0.00%	5.12%
11	92.16%	10.61%
12	85.23%	3.82%
	38.35%	8.39%

Table 2: Performance gains obtained on Apache web server.

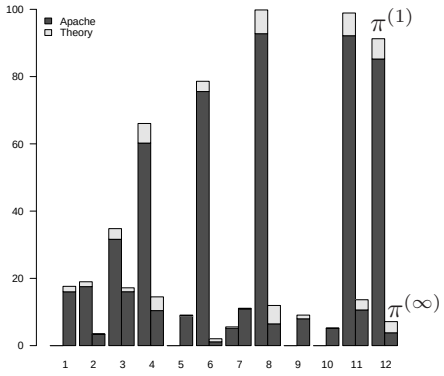

Figure 3: Comparison of gains: Apache vs. model

the response times, have a significant impact on the perceived performance. Alternatively, from the perspective of a service provider, the model can be extended to deal with multiple request types, each having its own service level agreement. These extensions raise many challenging questions that are of practical interest.

Second, in many web-based services a single user transaction induces a sequence of server and database requests. These requests do not need to progress through the system linearly, but may be routed in a general manner through the network, so that certain queues are visited more than once. In this case, the optimal decision rules may depend on the decision rules of all the queues, since requests that are behind a particular request can be routed such that they will be ahead the request. The insight provided by our model can prove to be useful for deriving optimal policies for this system.

Finally, many web server architectures deal with more than one shared resource at the hardware layer in addition to the CPU, e.g., memory, I/O, bandwidth, or more CPUs. Therefore, the highly distributed nature of today's information and communication infrastructures warrants research for multi-layered queueing models with multiple shared resources. Our model can serve as a basis for addressing issues related to the optimal allocation of hardware and software resources in these systems.

References

[1] van der Mei, R.D., Hariharan, R., Reeser, P.K.: Web server performance modeling. Telecommunication Systems **16** (2001) 361–378

[2] Harchol-Balter, M., Bansal, N., Schroeder, B.: Implementation of SRPT scheduling in web servers (2000)

[3] Harchol-Balter, M., Bansal, N., Schroeder, B., Agrawal, M.: SRPT scheduling for web servers. Lecture Notes in Computer Science **2221** (2001) 11–21

[4] Crovella, M.E., Frangioso, R., Harchol-Balter, M.: Connection scheduling in web servers. In: Proceedings USENIX symposium on Internet Technologies and Systems. (1999)

[5] Schrage, L.E.: The queue M/G/1 with the shortest remaining processing time discipline. Operation research **14** (1966) 670–684

[6] Hariharan, R., Ehrlich, W.K., Reeser, P.K., van der Mei, R.D.: Performance of web servers in a distributed computing environment. In de Souza, J.M., da Fonseca, N., de Souza e Silva, E., eds.: Teletraffic Engineering in the Internet Era. (2001) 137–148 also Proceedings 17th International Teletraffic Congress (Salvador, Dec. 2001).

[7] Rolia, J.A., Sevcik, K.C.: The method of layers. IEEE Transactions on Software Engineering **21** (1995) 689–699

[8] Woodside, C.M., Neilson, J.E., Petriu, D.C., Majumdar, S.: The stochastic rendezvous network model for the performance of synchronous client-server like distributed software. IEEE Transactions on Computers **44** (1995) 20–34

[9] Konheim, A., Meilijson, I., Melkman, A.: Processor-sharing of two parallel lines. Journal of Applied Probability **18** (1981) 952–956

[10] Cohen, J.W., Boxma, O.J.: Boundary value problems in queueing system analysis. North-Holland, Amsterdam (1983)

[11] Fayolle, G., Iasnogorodski, R.: Two coupled processors: the reduction to a Riemann-Hilbert problem. Zeitschrift für Warscheinlichkeitstheorie und Verwandte Gebiete **47** (1979) 325–351

[12] van der Weij, W., van der Mei, R.D.: Stability and throughput in a two-layered network of multi-server queues. In: Proceedings 3rd international working conference on Performance Modelling and Evaluation of Heterogeneous Networks, HETNETS. Number P02, Ilkley, England (2005)

[13] Schassberger, R.: Warteschlangen. Springer-Verlag (1973)

[14] Tijms, H.C.: Stochastic Models: An Algorithmic Approach. John Wiley & Sons, Chichester, England (1994)

[15] Asmussen, S., Nerman, O., Olsson, M.: Fitting phase type distributions via the EM algorithm. Scandinavian Journal of Statistics **23** (1996) 419–441

[16] Puterman, M.L.: Markov Decision Processes. John Wiley & Sons (1994)