Performance Evaluation 66 (2009) 301–310

Contents lists available at ScienceDirect

Performance Evaluation

journal homepage: www.elsevier.com/locate/peva

Dynamic thread assignment in web server performance optimization

Wemke van der Weij [a], Sandjai Bhulai [b,a,*], Rob van der Mei [a,b]

[a] CWI, Advanced Communication Networks, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
[b] VU University Amsterdam, Faculty of Sciences, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Received 23 January 2008
Received in revised form 5 October 2008
Accepted 19 November 2008
Available online 30 November 2008

Keywords:
Dynamic programming
Dynamic thread management
Multi-layered queueing systems
Web servers

ABSTRACT

Popular web sites are expected to handle huge number of requests concurrently within a reasonable time frame. 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 performance. Moreover, the commonly used policies do not take into account the complex interaction between the threads that compete for access to a shared 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 resource. This type of two-layered queueing model, which occurs naturally in the performance modeling of systems with intensive software–hardware interaction, are on the one hand appealing from an application point of view, but on the other hand are challenging from a methodological point of view. 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- and disk I/O-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 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. 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, see Fig. 1, an HTTP GET request may require processing in

* Corresponding address: Department of Mathematics, VU University Amsterdam, De Boelelaan, 1081a, 1081 HV Amsterdam, The Netherlands. Tel.: +31 6 46097077; fax: +31 20 5987653.
E-mail address: sbhulai@few.vu.nl (S. Bhulai).

0166-5316/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.peva.2008.11.001

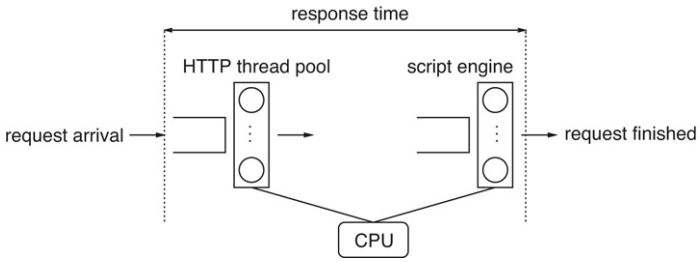

Fig. 1. A web server model.

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 [1,2].

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 fully understood (see [3] for recent results on software bottlenecks). 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 hardware (e.g., a CPU and disk) with a limited amount of 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 (PS) fashion; i.e., when there are k threads active at some moment in time, then each of these k threads receives a fair share $1/k$ of the total processor capacity [4]. 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, disk, memory) layer 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, the active threads effectively share the underlying resource: the more threads that are active, the smaller the processor capacity that 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, and highly challenging from a methodological point of view.

In this paper our objective is to construct a model that, on the one hand captures key features of web servers, but on the other hand is simple enough to allow for an analytic analysis. To this end, we make a number of simplifying assumptions: (1) each request traverses a fixed and known path, and (2) the active threads effectively share a single bottleneck hardware resource (CPU, disk) in a PS fashion (see also page 20 of [5]). Although these assumptions may not be entirely realistic in some practical situations (see also Section 5 for a discussion about the model assumptions and Section 6 for model extensions) the results presented in this paper provide important initial insights into the structure of optimal thread assignment policies for web servers (see also Section 6). In this context, the contribution of the paper should be viewed as a first important step to the development of thread policies for more complex web server models.

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. [6–8] and Crovella et al. [9] 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 model, the results show that the classical Shortest Remaining Processing Time (SRPT) policy is very effective [10]. In contrast to the present paper, it should be noted that the results in [6] 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. In this context, the contribution of the present paper complements the results obtained in the above references. Menascé [11] gives an overview of issues involved in modeling web servers. Cao et al. [12] propose to model a web server by a simple M/G/1/K/PS-queue, and validate the model through lab experiments. Detailed performance models for web servers, explicitly including the interaction between software and hardware contention, were proposed in [4,1]; these modeling efforts naturally led to the formulation of two-layered queueing models.

Several other papers also focus on queueing networks with a layered structure. Rolia and Sevcik [13] propose the Method of Layers (MoL), i.e., a closed queueing-network model based on the responsiveness of client–server applications. Woodside et al. [14] propose the so-called Stochastic Rendez-Vous Network (SRVN) model to analyze the performance of application software with client–server synchronization. Ramesh and Perros [15] model a web server system where clients and servers communicate via synchronous and asynchronous communication, and where the servers form a multi-layered hierarchical structure. They propose an approximate method for calculating the mean response time based on a decomposition approach. Dilley et al. [5] describe custom instrumentation to collect workload metrics and model parameters from large-scale web servers. They develop a layered queueing model (LQM) of a web server and use the model to predict the impact of a single web server thread pool size on the server and client response times. Franks et al. [3] focus on the correct definition and detection of bottlenecks in the context of layered queueing models. 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 [16–18]). For a two-layered network of two multi-server queues with processor sharing, remarkable results on the per-queue stability were obtained in [19].

Although a lot of progress has been made in understanding and improving the performance of web servers in the references outlined above, to the best of the authors' knowledge, the problem of *dynamic control* of threads in layered queueing networks has not been addressed in the literature.

In this paper, we model a web server by a two-layered queueing network with a single processor-shared 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 one 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.

The contribution of this paper is of both methodological and practical interest. First, from an application point of view, we derive explicit dynamic optimal thread-assignment policies for web servers, and show by experiments with an Apache web server that these polices indeed lead to significant reductions of the response times of the web server. Second, from a methodological point of view, the optimal thread-assignment policies derived in this paper are among the few exact detailed results for queueing networks with interacting servers; a class of queueing models for which hardly any exact results are known today. As such, the results derived in this paper can be seen as pioneering analytical contributions in the field of multi-layered queueing models. These observations make the contribution of this paper evident.

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. In Section 5 we discuss the model assumptions and the computational complexity of the thread-assignment policies. We conclude in Section 6 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 PS resource. For this purpose, consider a network of N queues in tandem with a common 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

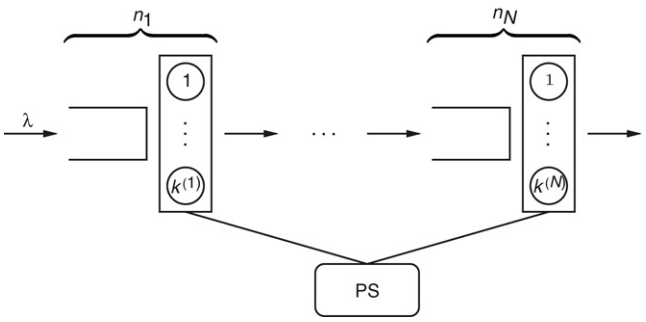

Fig. 2. A two-layered queueing system.

not explicitly model delays due to context switching between threads, since the CPU time in comparison to the processing times of the threads is negligible (see Remark 4.1 for a justification). The load on the system is given by $\rho = \lambda(\beta_1 + \cdots + \beta_N)$. This model is illustrated in Fig. 2.

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 with $M_i + 1$ states (where state $M_i + 1$ is the absorbing state), with initial distribution $\eta^{(i)} = (\eta_1^{(i)}, \ldots, \eta_{M_i}^{(i)})$, where $\eta_j^{(i)}$ is the probability that the Markov chain starts in state j for $j = 1, \ldots, M_i$. When the Markov chain is in state j, the time that the process spends in state j has an exponential distribution with parameter $\mu_j^{(i)}$. Upon leaving state j, the process jumps to state l with probability $p_{jl}^{(i)}$, or jumps to the absorbing state $M_i + 1$ with probability $p_{j,M_i+1}^{(i)}$, where $\sum_{l=1}^{M_i+1} p_{j,l}^{(i)} = 1$. The absorbing state corresponds to a completion of the service requirement at queue i.

Phase-type distributions have the important feature that they are dense in the class of all non-negative distributions, while retaining their tractability [20]. Therefore, it is possible to model heavy-tailed distributions by phase-type distributions. This is especially relevant, since it has been observed that file sizes on web servers follow a heavy-tailed distribution (see, e.g., [9]). 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., [21]), or by the more complex EM-algorithm (see [22]).

Next, we uniformize the system (see Section 11.5 of [23]). 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 Fig. 2, is then given by the tuple $(\vec{n}, \vec{k})$.

Let $\mathbb{E}W$ be the expected response time of an arbitrary customer in the system. The goal is to find a policy π^* that minimizes $\mathbb{E}W$. By using Little's Law (see [21]) this objective translates to minimizing the expected number of requests in the system. Therefore, we assume that 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 $g = \lim_{t\to\infty} u_t(x)/t$, i.e., the average number of requests in the system is independent of the initial state x (Proposition 8.2.1 of [23]). The expected response time $\mathbb{E}W$ can then be expressed as g/λ.

Let $V(\vec{n}, \vec{k})$ be a real-valued function defined on the state space. This function will play the role of the relative value function, i.e., 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) $g + V = TV$, where T is the dynamic programming operator acting on V defined as follows:

$$
\begin{aligned}
TV(\vec{n}, \vec{k}) = {} & \sum_{i=1}^{N}\sum_{j=1}^{M_i} n_j^{(i)} + \lambda\sum_{j=1}^{M_1}\eta_j^{(1)} H\left(\vec{n} + e_j^{(1)}, \vec{k}\right) + \sum_{i=1}^{N}\sum_{j=1}^{M_i}\sum_{l=1}^{M_i} \frac{k_j^{(i)}\mu_j^{(i)}p_{jl}^{(i)} H\left(\vec{n} - e_j^{(i)} + e_l^{(i)}, \vec{k} - e_j^{(i)} + e_l^{(i)}\right)}{k^{(1)} + \cdots + k^{(N)}} \\
& + \sum_{i=1}^{N}\sum_{j=1}^{M_i}\sum_{l=1}^{M_i} \frac{k_j^{(i)}\mu_j^{(i)}p_{j,M_i+1}^{(i)}\eta_l^{(i+1)}}{k^{(1)} + \cdots + k^{(N)}} H\left(\vec{n} - e_j^{(i)} + e_l^{(i+1)}, \vec{k} - e_j^{(i)}\right) \\
& + \left[1 - \lambda - \sum_{i=1}^{N}\sum_{j=1}^{M_i}\sum_{l=1}^{M_i+1} \frac{k_j^{(i)}\mu_j^{(i)}p_{jl}^{(i)}}{k^{(1)} + \cdots + k^{(N)}}\right] V(\vec{n}, \vec{k}),
\end{aligned}
$$

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

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

with $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$, where $V\left(\vec{n}, \vec{k} + ae_j^{(i)}\right)$ represents the action that spawns a additional threads at queue j. 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 leads to loss of capacity in the model. Therefore, the model ensures that threads will only be spawned for requests that are waiting for service.

A policy π maps the state $(\vec{n}, \vec{k})$ to an admissible action a, which represents the number of additional threads to be spawned in state $(\vec{n}, \vec{k})$. Thus, the information that one uses to derive optimal actions generally depends on the number of requests in each phase at each of the queues, as well as the number the number of outstanding threads for each request in each phase.

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 $k_j^{(i)} / \left(k^{(1)} + \cdots + k^{(N)}\right)$, since that request only uses a fair share of the service capacity. 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 requirement. 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.

The optimality equation $g + V = TV$ is hard to solve analytically in practice. Alternatively, the optimal actions can also be obtained 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 [23]). Consequently, when V is known, we can restrict our attention to the function H to obtain the optimal actions. In Section 4 we adopt this approach to compute optimal policies.

3. Dynamic thread management

In this section, we focus on dynamic thread assignment. 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. 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 $i > j$. This result also holds when the service-time distributions are replaced with Erlang distributions. This class of distribution 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 have a smaller expected sojourn time compared to a request in queue $j > i$ when a thread is opened at queue i. For this reason, the optimal policy can be obtained efficiently by recursive computation. As will follow from Theorem 3.1, 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 the expected sojourn time decreases monotonically as a request progresses to the last queue. 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 the expected sojourn time 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 obtain a smaller expected sojourn time than the expected sojourn time of a job further in the system 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 expected sojourn time of the s phase-j requests become smaller than the expected sojourn time of at least one job in queue $j \geq i$ under decision rules $\varphi_{i+1}, \ldots, \varphi_N$. Then policy $\pi = (\varphi_1, \ldots, \varphi_N)$ is optimal.*

Proof. The proof of Theorem 3.1 is fairly straightforward, but computationally cumbersome. Therefore, we only give a brief sketch of the proof. To this end, we first consider the case of exponential service times. In this case, the dynamic programming operator T becomes

$$TV(\vec{n}, \vec{k}) = \sum_{i=1}^{N} n_i + \lambda H(\vec{n} + e_1, \vec{k}) + \sum_{i=1}^{n} \frac{k_i}{K} \mu H(\vec{n} - e_i + e_{i+1}, \vec{k} - e_i) + \left(\mu - \sum_{i=1}^{N} \frac{k_i}{K} \mu\right) V(\vec{n}, \vec{k}),$$

with $K = k_1 + \cdots + k_N$. It turns out that $V(\vec{n}, \vec{k})$ satisfies the following properties (which can be proven by induction arguments on l in the relation $V_{l+1} = TV_l$ for arbitrary V_0):

(1) $V(\vec{n}, \vec{k}) \geq V(\vec{n} - e_i, \vec{k} - e_i)$,
(2) $V(\vec{n}, \vec{k}) \geq V(\vec{n} - e_i + e_{i+1}, \vec{k} - e_i)$,
(3) $V(\vec{n}, \vec{k}) \geq V(\vec{n}, e_s)$, $s = \max\{j | x_j > 0\}$.

The first inequality tells us that having fewer requests in the system is better. However, since requests can only leave the system when reaching queue N, we need the second inequality that states that having further requests in the system is better. Finally, these properties are needed to prove the last inequality which represents the optimal policy, i.e., serve requests in a FCFS manner. The case for general phase-type service distributions is proven with similar techniques. However, one needs to use additional properties (namely, submodularity and subconvexity) of $V(\vec{n}, \vec{k})$ in the proof, and the induction arguments become longer. Note that although the model is complex, the optimal policy is quite simple in many cases. Moreover, the policy can be implemented recursively, since the decision for requests at queue i depends only on the requests in queue $j \geq i$. Therefore, this model leads to an efficient implementation of dynamic thread-assignment policies in web server performance optimization.

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 $N = 2$, and where $M_i = 1$ or 2 for $i = 1, \ldots, N$. 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 $i > j$. 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, defined as

$$\left(\mathbb{E}W(\pi^{(s)}) - \mathbb{E}(\pi^*)\right) / \mathbb{E}W(\pi^*) \times 100\%,$$

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. In the other case, we mention both $\beta_1^{(i)}$ which is selected with probability $r_1^{(i)}$ and $\beta_2^{(i)}$ which is selected with probability $1 - r_1^{(i)}$. 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, as defined above. The last line in this table represents the average

Table 1
The performance of policies under twelve scenarios.

Case	$c_{s_1}^2$	$c_{s_2}^2$	$r_1^{(1)}$	$r_1^{(2)}$	$\beta_1^{(1)}$	$\beta_2^{(1)}$	$\beta_1^{(2)}$	$\beta_2^{(2)}$	$\pi^{(1)}$ (%)	$\pi^{(4)}$ (%)	$\pi^{(\infty)}$ (%)	$\pi^{(1,1)}$ (%)
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

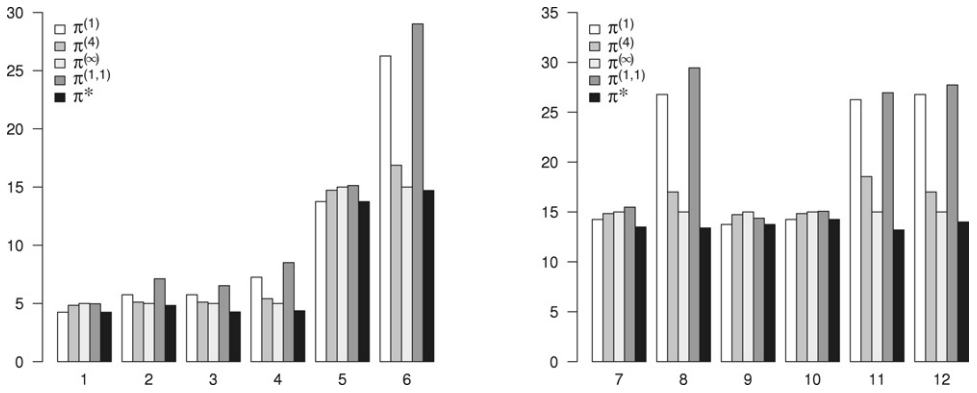

Fig. 3. Expected response times (in seconds) for the twelve scenarios.

gain compared to each policy. Note that the case with $\rho = 0.6$ is representative, since experiments with other loads give comparable performance gains.

Fig. 3 shows the expected response times in seconds for the five policies for the twelve scenarios. We can immediately see that the optimal policy π^* leads to significant reductions in the expected response times. 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 when compared to all other policies.

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 it 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 negligible, 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. The threads are spawned by the Apache web server itself, and consequently delays due to context switching between threads can affect the results 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

Table 2
Performance gains obtained on Apache web server for the twelve scenarios.

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

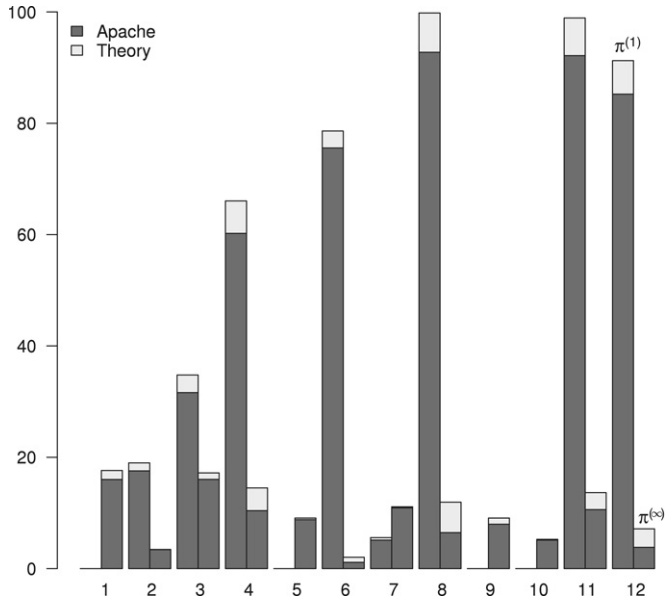

Fig. 4. Comparison of gains: Apache vs. model.

number of threads such that these phenomena do not occur in our experiments. In practice, the optimal policy bounds the number of threads so that the likelihood of causing other bottlenecks is limited.

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 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 Fig. 4 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. Note that the gains obtained by the model are generally higher than the gains obtained in the experimental setting. This can be explained by observing that context switching is not taken into account in the model. Moreover, we expect that the gains of π^* over $\pi^{(\infty)}$ strongly increase when the system is heavily loaded, since this will lead to superfluous context switching and hence a waste of processor capacity in the case of $\pi^{(\infty)}$.

5. Discussion

In this section we discuss the computational complexity of the optimal policy derived in Theorem 3.1 and discuss possible model extensions.

Remark 5.1 (*Computational Complexity*). The optimal policy of Theorem 3.1 is explicit for the case of exponentially and Erlang distributed service times. For other service distributions, the optimal policy can be computed efficiently by a recursive scheme starting with queue N and working backwards to queue 1. Thus, decision rule φ_i does not depend on the states $(n^{(j)}, k^{(j)})$ for $j < i$. This observation leads to a dramatic reduction in the number of states for which one needs to determine

the optimal decision rule. More precisely, the number of states that one needs to derive the decision rule for queue i is reduced from $\prod_{l=1}^{N} M_l$ to $\sum_{l=i}^{N} M_l$.

Remark 5.2 (*Feedback Control*). In this paper, it is assumed that the service-time distribution at each processing step is a known phase-type distribution, while in practice accurate measurements of the service-time distribution may not be trivial to obtain. In practice, custom instrumentation can be developed to estimate the workload (see [5]). Also, the processing-time distributions may change over time, which may impact the optimal thread-assignment policies. In practice, one can develop a feedback loop that re-estimates the service-time distribution regularly and adjusts the policies accordingly. The work in this paper forms the basis for this closed-loop control with feedback.

Remark 5.3 (*Multi-resource Possession*). In the present paper, it is assumed that the active threads share a single common hardware resource, modeled as a PS-server. Although this assumption works well in many cases (e.g., where the server is CPU- or I/O-bound), the model may be extended to include the fact that threads may occupy multiple resources simultaneously. In fact, 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, multi-cored processors or multiple CPUs). Therefore, the highly distributed nature of today's information and communication infrastructures warrants research for multi-layered queueing models with multiple shared resources. The derivation of efficient thread-assignment policies for models that allow threads to occupy multiple resources at the same time is far from trivial, and addresses an interesting topic for further research.

6. Conclusions and further research

We have considered the problem of dynamic thread assignment in web servers such that the expected response time is minimized. This problem can be translated into a Markov decision process problem for multi-layered queueing networks, a class of queueing networks for which hardly any exact detailed results have been obtained so far. We show that for phase-type service-time distributions, the optimal policy spawns a thread for a request if the resulting expected sojourn time of that request becomes smaller than the expected sojourn time, of at least one request further 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, 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 of the request. The insight provided by our model can prove to be useful for deriving optimal policies for this system.

Second, from a methodological point of view, it is challenging to investigate to what extent the results presented in this paper can be generalized to a more general class of multi-layered queueing models. For example, in many cases, the threads at the higher layer may not share an underlying resource that follows a processor-sharing discipline, but instead may be served, for example, on a first-come-first-served basis. Performance analysis and optimization for this type of model address a very challenging area for further research.

Third, another highly relevant extension of the model is to include communication with backend servers. In practice, Web servers usually operate in a multi-tiered environment in which dynamic content is gathered from downstream backend servers. In this context, an active thread that sends an information-retrieval request to a backend server communicates either synchronously or asynchronously. In case of synchronous communication, the thread is blocked while waiting for the requested information, while in the case of asynchronous communication, the thread proceeds to the next job and may be interrupted whenever the information retrieval is finished. Extension of the model to include communication with remote backend servers is an interesting topic for further research.

Finally, 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 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.

Acknowledgments

The authors would like to the thank the anonymous referees for their useful suggestions, which have led to significant improvements of the paper.

References

[1] R. Hariharan, W.K. Ehrlich, P.K. Reeser, R.D. van der Mei, Performance of web servers in a distributed computing environment, in: J. Moreira de Souza, N.L.S. da Fonseca, E.D. deSouza e Silva (Eds.) Teletraffic Engineering in the Internet Era, 2001, pp. 137–148. Proceedings 17th International Teletraffic Congress, Salvador, Dec. 2001.
[2] R.D. van der Mei, W.K. Ehrlich, P.K. Reeser, J.P. Francisco, A decision support system for tuning Web servers in distributed object-oriented network architectures, ACM Performance Evaluation Review 27 (2000) 57–62.
[3] G. Franks, D.C. Petriu, C.M. Woodside, J. Xu, P. Tregunno, Layered bottlenecks and their mitigation, in: Proc. of 3rd Int. Conference on Quantitative Evaluation of Systems, 2006, pp. 103–114.
[4] R.D. van der Mei, R. Hariharan, P.K. Reeser, Web server performance modeling, Telecommunication Systems 16 (2001) 361–378.
[5] J. Dilley, R. Friedrich, T. Jin, J. Rolia, Web server performance measurement and modeling techniques, Performance Evaluation 33 (1) (1998) 5–26.
[6] M. Harchol-Balter, B. Schroeder, N. Bansal, M. Agrawal, Size-based scheduling to improve web performance, ACM Transactions on Computer Systems 21 (2) (2003) 207–233.
[7] N. Bansal, M. Harchol-Balter, Analysis of SRPT scheduling: Investigating unfairness, in: Proceedings of ACM Sigmetrics Conference on Measurement and Modeling, Cambridge, MA, 2001, pp. 279–290.
[8] M. Harchol-Balter, N. Bansal, B. Schroeder, SRPT scheduling for web servers, in: 7th International Workshop, Job Scheduling Strategies for Parallel Processing, Cambridge, MA, in: Lecture Notes in Computer Science, vol. 2221, 2001, pp. 11–20.
[9] M.E. Crovella, R. Frangioso, M. Harchol-Balter, Connection scheduling in web servers, in: Proceedings USENIX symposium on Internet Technologies and Systems,1999.
[10] L.E. Schrage, The queue M/G/1 with the shortest remaining processing time discipline, Operation Research 14 (1966) 670–684.
[11] D.A. Menascé, Web performance modeling issues, International Journal of High Performance Computing Applications 14 (4) (2000) 292–303.
[12] J. Cao, M. Andersson, C. Nyberg, M. Kihl, Web server performance modeling using an M/G/1/K*PS queue, in: 10th International Conference on Telecommunications ICT 2003, vol. 2, 2003, pp. 1501–1506.
[13] J.A. Rolia, K.C. Sevcik, The method of layers, IEEE Transactions on Software Engineering 21 (1995) 689–699.
[14] C.M. Woodside, J.E. Neilson, D.C. Petriu, S. Majumdar, The stochastic rendezvous network model for the performance of synchronous client–server like distributed software, IEEE Transactions on Computers 44 (1995) 20–34.
[15] S. Ramesh, H.G. Perros, A multilayer client–server queueing network model with synchronous and asynchronous messages, IEEE Transactions on Software Engineering 26 (11) (2000) 1086–1100.
[16] A. Konheim, I. Meilijson, A. Melkman, Processor-sharing of two parallel lines, Journal of Applied Probability 18 (1981) 952–956.
[17] J.W. Cohen, O.J. Boxma, Boundary Value Problems in Queueing System Analysis, North-Holland, Amsterdam, 1983.
[18] G. Fayolle, R. Iasnogorodski, Two coupled processors: The reduction to a Riemann–Hilbert problem, Zeitschrift für Warscheinlichkeitstheorie und Verwandte Gebiete 47 (1979) 325–351.
[19] W. van der Weij, R.D. van der Mei, 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, in: HETNETS, vol. P02, Ilkley, England, 2005.
[20] R. Schassberger, Warteschlangen, Springer-Verlag, 1973.
[21] H.C. Tijms, Stochastic Models: An Algorithmic Approach, John Wiley & Sons, Chichester, England, 1994.
[22] S. Asmussen, O. Nerman, M. Olsson, Fitting phase type distributions via the EM algorithm, Scandinavian Journal of Statistics 23 (1996) 419–441.
[23] M.L. Puterman, Markov Decision Processes, John Wiley & Sons, 1994.

Wemke van der Weij (1982) received her B.Sc. degree in Econometrics and Operations Research and her M.Sc. in Operations Research, both from the University of Amsterdam. In 2005 she started a Ph.D. program at the Center for Mathematics and Computer Science (CWI), also being a member of the Optimization of Business Processes group at the VU University, Amsterdam. Her research interests are performance analysis of stochastic networks, and in particular the analysis of queueing networks with shared resources. She is expecting to defend her Ph.D. dissertation on spring 2009.

Sandjai Bhulai (1976) received his M.Sc. degrees in Mathematics and in Business Mathematics and Informatics, both from the VU University Amsterdam, The Netherlands. He carried out his Ph.D. research on "Markov decision processes: the control of high-dimensional systems" at the same university for which he received his Ph.D. degree in 2002. After that he has been a postdoctoral researcher at Lucent Technologies, Bell Laboratories as NWO Talent Stipend fellow. In 2003 he joined the Mathematics department at the VU University Amsterdam, where he is an assistant professor in Applied Probability and Operations Research. His primary research interests are in the general area of stochastic modeling and optimization, in particular, the theory and applications of Markov decision processes. His favorite application areas include telecommunication networks and call centers. He is currently involved in the control of time-varying systems, partial information models, dynamic programming value functions, and reinforcement learning.

Rob van der Mei (1966) received his M.Sc. degrees in Mathematics and in Econometrics, both from the VU University Amsterdam. In 1995 he received his Ph.D. degree from University of Tilburg, The Netherlands. After that he has been working as a consultant and researcher in the telecommunication industry, for The Royal Dutch PTT, AT&T Labs and TNO for over a decade. In 2004 he joined the Centre for Mathematics and Computer Science (CWI), where he is currently the head of the Department of Probability and Stochastic Networks, and the leader of the research theme Societal Logistics. He also has a part-time assignment as a full professor at the VU University, Amsterdam, The Netherlands, where he is responsible for the research and education in the field of communication networks, with a particular focus on performance aspects. He is a member of the editorial board of Performance Evaluation and the AEUE Journal on Electronics and Communications. His research interests include performance analysis of communication networks, health care logistics and queueing theory, and more recently, he has started several projects in Revenue Management, Grid computing and sensor networks. He is teaching a variety of industry-oriented courses on Performance Management and Design of ICT systems for system architects, and is consulting for ICT companies on a regular basis. He has published over 80 refereed papers in the field.