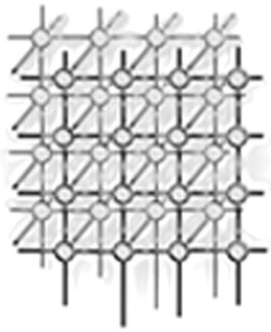
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **22**:2421–2444 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



Optimizing utilization of resource pools in web application servers

Alexander Totok[1,*,†] and Vijay Karamcheti[2]

[1] *Google Inc., New York, NY, USA*
[2] *Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, NY, USA*

SUMMARY

Among the web application server resources, most critical for its performance are those that are *held exclusively by a service request* for the duration of its execution (or some significant part of it). Such exclusively-held server resources become *performance bottleneck points*, with failures to obtain such a resource constituting a major portion of request rejections under server overload conditions. In this paper, we propose a methodology that computes the optimal pool sizes for two such critical resources: *web server threads* and *database connections*. Our methodology uses information about incoming request flow and about fine-grained server resource utilization by service requests of different types, obtained through offline and online request profiling. In our methodology, we advocate (and show its benefits) the use of a database connection pooling mechanism that caches database connections for the duration of a service request execution (so-called *request-wide database connection caching*). We evaluate our methodology by testing it on the TPC-W web application. Our method is able to accurately compute the optimal number of server threads and database connections, and the value of sustainable request throughput computed by the method always lies within a 5% margin of the actual value determined experimentally.

KEY WORDS: Internet services; web application server performance; database connection caching; provisioning and optimization of server resource pools; mathematical modeling

1. Introduction

1.1. Motivation

Modern Internet services such as e-mail, banking, online shopping, and entertainment web sites have become commonplace in recent decade. Providers of these services often implement them as applications hosted on web application (middleware) servers, such as IBM WebSphere [1], Oracle WebLogic [2], and Red Hat JBoss [3]. One of the typical situations that service providers (system administrators) have to deal with operating Internet services is the problem of degraded service

*Correspondence to: Alexander Totok, Google Inc., 76 9th Ave, 6th Floor, New York, NY 10011, USA
†E-mail: totok@google.com

Received 2 July 2009
Revised 16 January 2010
Accepted 30 January 2010

performance due to *server overload.* One of the approaches to improve application performance in the situation of high load or overload is to use various server-side resource management mechanisms to improve utilization of server resources [4, 5, 6].

Modern web middleware platforms are complex software systems that expose to system administrators several mechanisms that can be independently tuned to improve Internet application performance and optimize server resource utilization. The middleware layer itself rarely has control over low-level OS mechanisms, such as CPU scheduling and memory management. Instead, it provides control over higher-level resources, such as threads, database connections, component containers, etc. Some of these resources can be shared among concurrent user requests, but some are *held exclusively by a request* for the duration of its execution (or some significant part of it). Therefore, such non-shared server resources become *bottleneck points*, and failure to obtain such a resource constitutes a major portion of request rejections under high load or overload conditions [7]. This paper focuses on optimizing utilization of web application server resources that are held exclusively by a service request.

The most important of such exclusively held server resources in web middleware platforms are *server threads* and *database (DB) connections*. The application server creates and pools a limited (predefined) number of threads and database connections and schedules them to the incoming user requests. The pooling mechanism avoids expensive operations for creating and closing of these server resources. One of the questions that system administrators face in this context is *what is the optimal number of threads and database connections* that achieve the highest request throughput? Using more threads and database connections allows for increased execution parallelism, but may result in degraded performance due to thread context switching and increased data and locking contention in the database. The task of identifying the optimal number of threads and database connections is further complicated by the fact that for different user loads, different configurations of the thread and database connection pools provide the optimal application performance. This happens because different sets of application components and middleware services are used to execute *requests of different types*. Some requests, for example, need to access a database (so they need to obtain a database connection), while some don't.

1.2. Methodology

In this paper, first, we propose a model for request execution with 2-tier exclusive resource holding (1st tier: threads, 2nd tier: database connections). In this model, we advocate (and later show its benefits) the use of a database connection pooling mechanism that caches database connections for the duration of the service request execution. This approach (which we call *request-wide database connection caching*) contrasts with the standard *transaction-wide database connection caching* employed by the majority of modern web application servers.

Second, we propose a methodology that computes the optimal number of threads and database connections for a given Internet application, its server and database environment, and specific user load (request mix). The methodology is built on the proposed model for request execution with 2-tier exclusive resource holding. The methodology works as follows.

- First, a limited set of offline experiments are conducted, where the actual application (Internet service) and its server environment are subjected to an artificial user load. We use a different number of threads and database connections for each test run, and only a subset of possible values for these resource pools is used throughout the experiments. During this series of "profiling tests", information about *fine-grained server resource utilization* is obtained through the instrumented profiling of request execution. More specifically, we are interested in the times spent by the requests of different types in the different stages of request processing.

- Second, the obtained values for these timing parameters (considered as functions of the number of threads and database connections) are used as data points for *function interpolation* to get the values of these parameters for all possible combinations of the number of threads and database connections.
- Third, under real operating conditions, the proposed request execution model takes as input these interpolated functions and the information about the actual *request flow* (request mix), obtained through online request profiling, and computes the number of threads and database connections that provide the best request (session) throughput (which is also computed by the method), thus achieving optimal utilization of web server threads and database connections. If the user load changes, the optimal number of threads and database connections can be recomputed using the new values of incoming request mix, thus enabling the possibility of *dynamic adaptation* of web application server environment to changing user load conditions. The computed value of maximum sustainable session throughput can be used to compute the number of servers required to support the user load and to (dynamically) allocate new server resources if there is a need.

We have implemented the proposed technique of request-wide database connection caching and aforementioned request profiling mechanisms as middleware services, which are seamlessly and modularly integrated in the open-source Java EE [8] application server JBoss [3]. The request profiling module performs automatic real-time monitoring of incoming client requests to extract parameters of service usage (i.e., incoming request mix). It also performs fine-grained request execution profiling to identify the times spent by requests of different types in different stages of their processing. The collected information is used later in the proposed request execution model to compute the optimal number of server threads and database connections.

We evaluate our approach on the TPC-W benchmark application [9], emulating a typical Internet application, an online store selling books. We use our method to compute the optimal number of threads and database connections and the value of maximum sustainable session throughput. We also determine these values experimentally, trying different sizes of the thread and database connection pools. Our results show that the proposed method is always able to accurately compute the optimal number of threads and database connections, and the value of maximum sustainable session throughput computed by the method always lies within a 5% margin of the actual value determined experimentally.

The rest of the paper is organized as follows. Section 2 provides necessary background. Section 3 presents the proposed technique of request-wide database connection caching and the request execution model with 2-level exclusive resource holding, which forms the basis of the method for computing the optimal number of web application server threads and database connections (presented in Section 4). In Section 5 presents our evaluation methodology and experimental results. In Section 6 we discuss related and future work, and we conclude in Section 7.

2. Background

2.1. Web application server architecture

In this paper, we study Internet services implemented on top of modern middleware platforms, such as the Java EE component framework [8]. Such services are usually built as complex software systems, consisting of several logical and physical tiers (e.g., web tier, application tier, and database tier) and accessing backend data sources. In this study, we don't differentiate between logical application tiers (e.g., web tier and application tier), as long as they belong to a single physical tier (e.g., single web

application server). In such view, an Internet application would have two tiers: (1) *web/application tier* running on a web application server, where requests are received, processed, and response is generated; and (2) *database tier*, represented by a database server, where application persistent data is stored and where the requests/updates to this data (issued by the web/application tier) are processed. We present our techniques in this rather centralized setting, to focus on the essence and benefits of the proposed method to compute the optimal size of exclusively-held resource pools (e.g., threads and database connections). We believe that the method will show its utility in a distributed setting as well, where it can be independently applied at every web/application/database server that sees concurrent requests competing for server resources that are held exclusively by the executing requests.

For scalability and enhanced performance reasons, modern web application servers usually adopt the mechanism of *resource pooling* for resources that are expensive to create and destroy, such as server threads and database connections. For example, in the case of JDBC [10] database connections, when an application needs to access the database, it performs a lookup for a special adaptor object, which, upon request, supplies the application with a wrapper object containing the actual JDBC database connection taken from the pool. When the application code "closes" the connection, it does so with the wrapper object, which in turn does not close the actual physical database connection, but rather returns it to the pool. This pooling mechanism allows one to avoid expensive operations of creating and closing database connections. In Section 3 we give more detailed description of the request execution model adopted in this study.

2.2. User load structure

Typical interaction of users with modern Internet services is organized into *sessions*, a sequence of related requests coming from a single user, which together achieve a higher level user goal. An example of such interaction is an online shopping scenario for an e-Commerce web site, which involves multiple requests that search for particular products, retrieve information about a specific item (e.g., quantity and price), add it to the shopping cart, initiate the check-out process, and finally commit the order. Collectively, requests issued by clients of an Internet service result in an incoming service request flow with certain characteristics. Different request flows produce different loads on various server resources, because requests of different types, when executed, use different sets of application components and server resources. Some requests, for example, need to access a database (so they need to obtain a database connection), while some don't.

The proposed method to compute the optimal number of threads and database connections requires as input the information about *user load structure*. Specifically, it needs to know the *proportion of requests of different types in the overall incoming request flow*. We capture this information by introducing the following parameters of user load structure: V_i — average number of requests of type i in a user session, and L – average user session length (in requests). The proportion of requests of type i in the overall request flow is then computed as V_i/L.

2.3. TPC-W application

To test the proposed method to compute the optimal values of critical resource pools, we use the TPC-W transactional web e-Commerce benchmark [9], that emulates an online store that sells books. The TPC-W specification describes in detail the application data structure and the 14 web invocations that constitute the web site functionality, and defines how they change the application data stored in the database. A typical TPC-W user session consists of the following requests: a user starts web site navigation by accessing the Home page, searches for particular products (Search), retrieves

information about specific items (Item Details), adds some of them to the shopping cart (Add To Cart), initiates the check-out process, registering and logging in as necessary (Register, Buy Request), and finally commits the order (Buy Confirm).

The TPC-W specification describes in detail the application data that should populate the database. The database population is defined by two parameters: NUM_ITEMS — the cardinality of the ITEM table, and NUM_EBS — the number of concurrent *Emulated Browsers (EB's)*, i.e., it is defined by the size of the store and size of the supported customer population. The size of the TPC-W database tables are fixed or depends linearly on the above two parameters. Application performance (i.e., the speed of processing database SQL queries) depends directly on the size of the database population.

We use our own implementation of the TPC-W benchmark, realized as a Java EE component-based application [11]. The implementation utilizes the *Session Façade* design pattern [12]. For each type of service request there is a separate Java servlet [13] which, if necessary to generate the response HTML page, invokes business method(s) on associated session EJB [14] application component, that in turn access application shared data stored in the database through a set of fine-grained invocations to the related entity EJB application components.

3. Request execution and database connection caching

In this study, we adopt the following model of request execution by the web application server. Requests compete for two critical exclusively-held server resources: server threads and database connections; these resources are pooled by the web application server. If the timeout for obtaining a thread or a database connection expires, the request is rejected with an explicit rejection message (note that some requests do not require database access, so they can be successfully served just by acquiring a server thread). This approach is shared by a vast majority of robust server architectures that bound request processing time in various ways (e.g., by setting a deadline for request completion), as opposed to a less robust approach, where a request is kept in the system indefinitely, until it is served (or is rejected by lower-level mechanisms such as TCP timeout). Our approach has the advantage of more efficiently freeing up server resources held by requests whose processing cannot be completed because of server capacity limitations. An important feature of our request execution model is that the database connection obtained by a request *remains available for exclusive use by the request until the request is processed*. After that the thread and the database connection(s) cached by it are returned to their respective pools.

The rationale for caching database connections for the duration of request execution is as follows. To access a database a request obtains a database connection from the database connection pool. After the required work (i.e., communication with the database) is done over this database connection, the latter is returned to the pool. The particulars of the Java EE platform are such that a database connection may be requested from the pool (and returned there) up to several dozen times during the execution of a single service request. For example, each business method invoked on an entity EJB [14] application component usually requires synchronization with the database (before or after the method invocation, or both). This results in a database connection being requested (and returned) from the pool up to three times just during one EJB method invocation (if the synchronization is performed before the EJB method invocation, the additional EJB-specific `ejbFindByPrimaryKey()` method accounts for the third request [14]).

Most JDBC [10] drivers additionally require that all database accesses on behalf of a single database transaction be performed over a single JDBC database connection (to ease the implementation of transaction rollback and commit). To implement this requirement, application servers, while

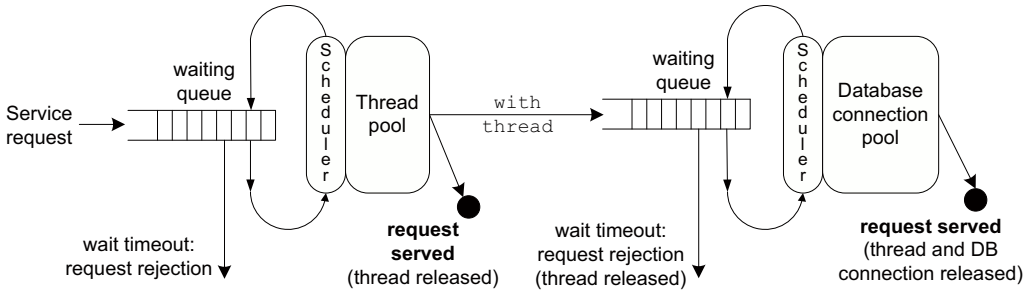


Figure 1. Request execution model with 2-level exclusive resource holding.

performing database connection pooling, cache database connections for active transactions and return them to the pool only after the transaction completes. This approach, of course, reduces the number of times a database connection is redundantly revoked and returned to the pool during the execution of a single request, but only for the requests associated with a transactional context. However, it is a common practice for Internet application developers to implement as few transactional service requests as possible, and usually only those that update back-end databases. This approach greatly decreases locking contention in the database and improves application performance. Given that modern Internet services typically exhibit dominant non-transactional *read-only* data access patterns, the majority of service requests cannot benefit from the approach of transaction-wide caching of database connections.

Similarly, in the situation of server overload, when database connections become a scarce resource, concurrent requests compete for database connections, experiencing queueing delays while trying to get them from the pool. The application performance can potentially deteriorate, because the same request has to get a connection from the pool several times. It not only increases request processing times, but also aggravates the situation by making requests that wait for a database connection waste other exclusively held resources, such as server threads.

3.1. Request-wide database connection caching

To alleviate this problem, we propose a database connection pooling mechanism that **caches database connections for the duration of the service request execution**. This mechanism, in addition to caching database connections for active transactions, caches at least one database connection for each request that has requested a database connection previously. Note that this does not guarantee that all the database activities for a single request can be performed over a single database connection. If a request accesses two databases, or starts two transactions, or interleaves non-transactional activities with transactional ones, then these communications with the database(s) need to be performed over different database connections. For the purposes of this study we however make a simplifying assumption that there is a single database that stores the application data and that all the communication with the database, required to process a single service request, can be made over a single database connection. Fig. 1 schematically illustrates the 2-level model of request execution and the flow of a request through the system, with the adopted **request-wide database connection caching** (as opposed to the standard transaction-wide database connection caching).

Of course, this approach of request-wide database connection caching may have its own drawbacks. Imagine, that between periods of communication with the database, a request performs some CPU-intensive processing of the application data. Returning the cached database connection to the pool in this situation would allow some other requests to progress and would increase parallelism. While possible, we feel that this concern is not as relevant for the heavy database-centric applications that this work targets, such as TPC-W, where there is little data manipulation between accesses of the database.

4. Computing the optimal number of threads and database connections

The goal of the method proposed in this study is to compute the number of threads M and the number of database connections N ($M \geq N$), that would maximize session throughput, for a given incoming request mix. The information about the latter comes in the specification of V_i, average number of requests of type i in a session (Section 2.2). The method works with the assumption that the underlying hardware and middleware environments, as well as application configuration parameters are fixed.

In our 2-level model of request processing with request-wide database connection caching (Section 3), request execution time can be represented as follows:

$$t = w^{\mathrm{THR}} + p + w^{\mathrm{DB}} + q \tag{1}$$

where w^{THR} is the time spent by the request in waiting for a thread, p is the time the request spends on processing before getting a database connection, w^{DB} is the time spent by the request in waiting for a database connection, and q is the time the request spends processing with a database connection in its possession. The latter includes the time spent in making SQL queries to the database, retrieving the results, processing them, and other request processing while the database connection is cached by the request. In this study, we treat the database as a black box and do not track database activities performed over database connections. Note that for requests that do not access the database, $w^{\mathrm{DB}} = q = 0$.

The maximum sustainable session throughput depends on the values of p and q, which are different for different request types. The main assumption we make is that under the maximum server load, $p_i(M, N)$ and $q_i(M, N)$ — the average times spent processing requests of type i before obtaining a database connection and with a database connection, respectively — **depend only on M and N, and on the structure of incoming request mix** (the values of V_i).[†] We also assume that **the dependence of $p_i(M, N)$ and $q_i(M, N)$ on the incoming request mix (V_i) is very weak**, that is, the values of $p_i(M, N)$ and $q_i(M, N)$ change insignificantly when the request load parameters V_i stay *close enough* to their initial values. These assumptions are justified by the results of the experiments that we conducted while working on this problem.

In an optimal server configuration we would want to achieve a balanced utilization of server threads and database connections. This means that under maximum sustained user load, we would want all threads and database connections to be fully utilized (they may be idle for some short periods of time, due to inevitable request burstiness, but the ideal situation is that all threads and database connections are always "busy" processing requests). Having a noticeable number of idle threads or idle database connections in situations where database connections or threads (respectively) become the resource bottleneck is a waste of server resources and will cause performance degradation.

[†]Of course, they depend on the application configuration, the hardware and the middleware, but those are fixed.

Assume for now, that we know the functions $p_i(M,N)$ and $q_i(M,N)$. With only N database connections we can not process, on average, more than

$$\lambda_{\mathrm{DB}}(M,N) = \frac{N}{\sum_i V_i q_i} \tag{2}$$

sessions per unit time, because incoming sessions have a certain number of requests that require database processing. For the same reason, with only M threads, we can not process more than

$$\lambda_{\mathrm{THR}}(M,N) = \frac{M}{\sum_i V_i (p_i + q_i)} \tag{3}$$

sessions per unit time, and this value is achievable if requests don't wait for database connections (i.e., $w_i^{\mathrm{DB}} = 0$). If we measure the throughput in number of sessions processed per unit time, than the maximum sustainable session throughput is given by the equation:

$$\lambda(M,N) = \min\{\lambda_{\mathrm{DB}}(M,N), \lambda_{\mathrm{THR}}(M,N)\}. \tag{4}$$

The best configuration of server resource pools (i.e., the values of M and N) is the one, which maximizes the value in equation (4). The value in equation (4) has a global maximum *inside* a bounded region of possible values of M and N. Indeed, $\lambda_{\mathrm{DB}}(M,M) > \lambda_{\mathrm{THR}}(M,M)$, since if numbers of threads and database connections are equal, threads present the "scarce" resource, because some requests do not require database access and database connections may have idle periods. In comparison, $\lambda_{\mathrm{DB}}(M,1) \leq \lambda_{\mathrm{THR}}(M,1)$ (i.e., database connections are the "scarce" resource), when M is big enough, because one database connection can only do a limited amount of work. With M and N growing, the performance of the server deteriorates, and the values of both $\lambda_{\mathrm{DB}}(M,N)$ and $\lambda_{\mathrm{THR}}(M,N)$ go down. The situation when values of $\lambda_{\mathrm{DB}}(M,N)$ and $\lambda_{\mathrm{THR}}(M,N)$ are equal represents an optimal correspondence of the number of threads to the number of database connections, which is a desirable situation. If sessions are coming with the rate of $\lambda_{\mathrm{DB}}(M,N) = \lambda_{\mathrm{THR}}(M,N)$ and if there is no burstiness in the request arrival pattern, then in an ideal processing environment all the requests will get served, with all the threads and database connections being constantly busy processing the incoming requests.

These considerations lead us to the method, which consists of several steps schematically shown in Fig 2.

Step 1. Because of the assumptions we made earlier in this Section, we may consider $p_i(M,N)$ and $q_i(M,N)$ as two-dimensional functions, defined for the triangular grid of integer arguments (M,N), $M \geq N > 0$. The goal of this step is to obtain the values of p_i and q_i for some subset of possible values of M and N. These data points will be used for the interpolation of functions $p_i(M,N)$ and $q_i(M,N)$ on their domain. We choose the data points as a sub-grid of the functions' domain, for example: $M, N = 1, 5, 10, 15, \ldots$ $(M \geq N)$.

To obtain the values of p_i and q_i, for each interpolation point (M,N) we subject the actual server environment with an artificially induced user load slightly surpassing the server capacity. For this series of "profiling tests" we choose the parameters of the request mix (V_i) representative of the actual user load, or close to the load that we expect the system will see during its real-life operation. Due to the server overload conditions, either just one of the two, or both pooled server resources (threads and database connections) will be used to their full capacity, and request queues will build up for threads, or database connections, or both. However, if the overload is not too big, this overflow will be handled by the server in a graceful manner, because of our request processing mechanism that explicitly rejects requests that can not obtain a thread or a database connection within a predefined time interval. During these tests we obtain the values of $p_i(M,N)$ and $q_i(M,N)$ through fine-grained profiling of request

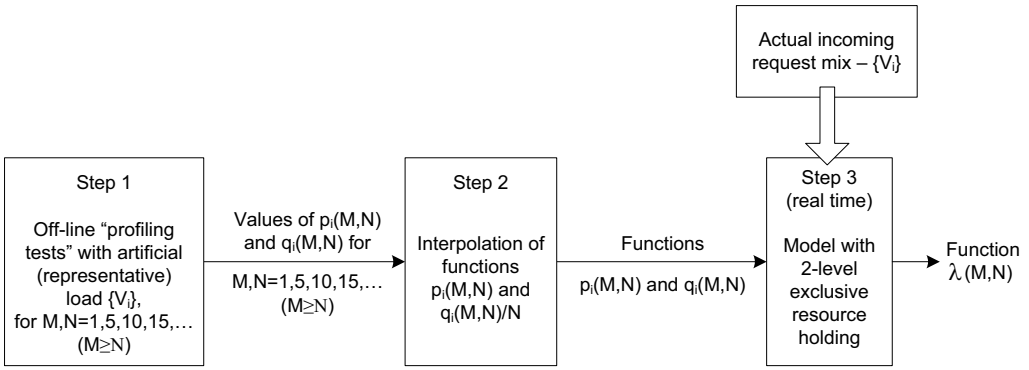


Figure 2. Logical steps of the method to compute the optimal number of server threads and database connections.

execution in the web application server. Aborting some requests (and so — some sessions) will alter a bit the mix of served requests (as compared with the mix of requests *submitted to the system*), but as we stated earlier, the dependance of p_i and q_i on the request mix is very weak, so the measurements will produce very close to the correct values.

Step 2. The obtained data points for functions $p_i(M, N)$ and $q_i(M, N)$ are used for the interpolation of these functions on their domain ($M \geq N > 0$). To get a smooth interpolation we use the method of two-dimensional piecewise bi-cubic interpolation [15]. Instead of interpolating function $q_i(M, N)$ we interpolate function $q_i(M, N)/N$, which turns out to be smoother and easier to interpolate than $q_i(M, N)$. It also has a meaningful interpretation: it is the inverse of the database throughput seen by requests of type i. We refer the reader to Section 5.3 for examples of interpolated functions $p_i(M, N)$ and $q_i(M, N)$.

Step 3. This step, unlike the first two ones, is performed in operating conditions and in real time, when the server environment is subjected to actual user load. The parameters of incoming request mix — the values of V_i — are obtained through online request profiling. As long as they stay *close enough* to the values used in the preliminary "profiling tests" (step 1), this method can be used to compute the optimal number of threads and database connections. Substituting obtained values V_i and the values of $p_i(M, N)$ and $q_i(M, N)$, obtained in step 2, into the equations (2), (3), and (4), we get the value of maximum sustainable session throughput $\lambda(M, N)$, for every possible combination of M and N. Given that parameters M (the number of server threads) and N (the number of database connections) are discrete and have a limited (and rather small) set of possible values, it is possible to iterate over this set in order to, first, determine the optimal number of threads for a given number of database connections, and vice versa, and second, find the pair of parameters (M, N) that achieve the highest session throughput.

The method computes the number of threads and database connections that provide the best session throughput (which is also computed by the method), thus achieving optimal utilization of web server threads and database connections. If the user load changes, the optimal number of threads and database connections can be recomputed (step 3) using the new values of incoming request mix (the values of V_i), thus enabling the possibility of dynamic adaptation of web application server environment to changing user load conditions. Using the observed (or anticipated) rate of new session arrival, the

computed value of maximum sustainable session throughput can be used to compute the number of servers required to support the user load and to (dynamically) allocate new server resources if there is a need.

Note that if the actual user load deviates considerably from the "profiling" load (used in step 1), then it may not be possible anymore to use in step 3 the values of $p_i(M, N)$ and $q_i(M, N)$, obtained in step 1 and step 2. In such situation, the profiling tests (step 1) should be rerun with the newer user load and new values of $p_i(M, N)$ and $q_i(M, N)$ should be obtained.

5. Experimental evaluation

In this section, we evaluate the costs and benefits of the proposed request-wide caching mechanism for database connections (Section 3.1). Then we present the evaluation of the proposed method for computing the optimal number of threads and database connections (Section 4). Before proceeding to the evaluation of the proposed techniques, we first present our experimental setup.

5.1. Experimental setup

Server configuration. As was stated earlier, we present our method to compute the optimal size of critical server resource pools in a simplified centralized setting in order to focus on the essence and benefits of the proposed method. Our experimental infrastructure consists of a web application server and a separate database server, each running on a dedicated workstation, connected by a high-speed local-area network. We use Java EE application server JBoss [3] (augmented with Jetty [16] HTTP/web server) as a web application server and MySQL database [17] for the database server. A separate workstation is used to produce user load.

Request profiling infrastructure. Our JBoss/Jetty web application server is augmented with the request profiling infrastructure [7, 18], implemented in a modular fashion as several pluggable middleware services, with some additional modifications made to the original code of JBoss/Jetty web application server and MySQL JDBC driver. The Jetty HTTP/web server is used to gather information about incoming client requests, which are classified by their type (based on the URL pattern) and session affiliation. Various JBoss modules, such as the Database Connection Manager, are augmented with execution hooks to gather information about the breakdown of request processing times spent at different request execution phases (e.g., waiting for a thread, processing in the database, etc.). When a request completes, the information about its execution is added to a server-wide request profiling in-memory store. The information gathered by the request profiling process is used then to compute parameters V_i, L, p_i, and q_i.

Web workload model. In this study, as in numerous other web server performance studies, we use synthetic web workloads, which are injected into a working application server environment using a load generator machine. Utilizing synthetic workloads is a common and widely adopted way to evaluate web server performance [19, 20, 21]. Although not as realistic as using real web traces, this approach is more convenient for controlled exploration of the range of client behaviors. A dominant fraction of web workload models [22, 23], as well as workload generators of web server performance benchmarks, such as TPC-W [9], use first-order Markov chains to model user session structure. Our study follows this established practice. We model session inter-request user think times as exponentially distributed with mean 10s. The flow of incoming new user sessions is modeled as a Poisson process with arrival rate λ, which determines the average request rate received by the system: $\lambda \cdot L$, where L is the average session length, in requests.

Table I. The values of p_i and q_i (in ms) for the TPC-W application, in the underloaded and "max-loaded" server environments ($M = N = 30$).

Request type	p_i underloaded	q_i underloaded	p_i "max-loaded"	q_i "max-loaded"
Home	97	0	235	0
Search	46	50	101	646
Item	9	9	26	311
Add To Cart	24	5	61	227
Cart	2	0	8	0
Register	2	0	7	0
Buy Request	94	59	205	940
Buy Confirm	93	112	230	1165

TPC-W configuration. In the standard TPC-W configuration with a typical database population, the performance bottleneck of our server environment is always the MySQL database server. In order to evaluate our model, we need an application that would equally stress the application server and the database server. To achieve this, we choose TPC-W configuration parameters and make some changes to the TPC-W application code to make the application less database-centric and remove request processing focus from SQL query processing. First, we use the smallest database population size: $\mathrm{NUM\_EBS} = 1$, $\mathrm{NUM\_ITEMS} = 100$ (see Section 2.3 for description of these parameters and TPC-W service request types). Second, we use in-memory (HEAP) database tables in the MySQL database, which further speeds up SQL query processing. Third, we remove presentation of randomly chosen advertisements from the Home page, so that this request now does not require database access. And finally, we insert into the application code CPU-consuming code snippets, which are designed to imitate some CPU-intensive application server processing (for example, SSL processing) *before* the request obtains a database connection (so it increases the values of p_i). These artificial code snippets produce different load for different request types (measured in *execution cycles*; each execution cycle is approximately 1 ms of execution time on our server environment, if request is executed in isolation): Home, Buy Request, Buy Confirm: 100 cycles, Search: 50, Add To Cart: 25, Item: 10, Cart, Register: 1. Table I shows p_i and q_i for the underloaded server (when only one request is executed at a time), and for the server stressed to its maximum load capacity ("max-loaded" server), with $M = N = 30$. It shows how p_i and q_i change when the load on the server increases.

5.2. Costs and benefits of the request-wide database connection caching

In this section we evaluate the proposed mechanism for request-wide caching of database connections (Section 3.1) by comparing its performance with the performance of the default transaction-wide database connection caching. We test our TPC-W application with various levels of user load and measure the times that requests spend in different stages of their execution. For the sake of the space, we present the results of experiments with fixed values of M and N (number of threads and database connections): $M = N = 30$. Experiments with different values of M and N show relative performance similar to the results shown below.

Figs. 3, 4, and 5 show the breakdown of request processing times for the Search, Buy Request, and Buy Confirm TPC-W requests respectively, for various levels of user load measured in percentage of server capacity. The left columns show results for our request-wide database connection caching mechanism, while the right columns show results for the default transaction-wide database connection

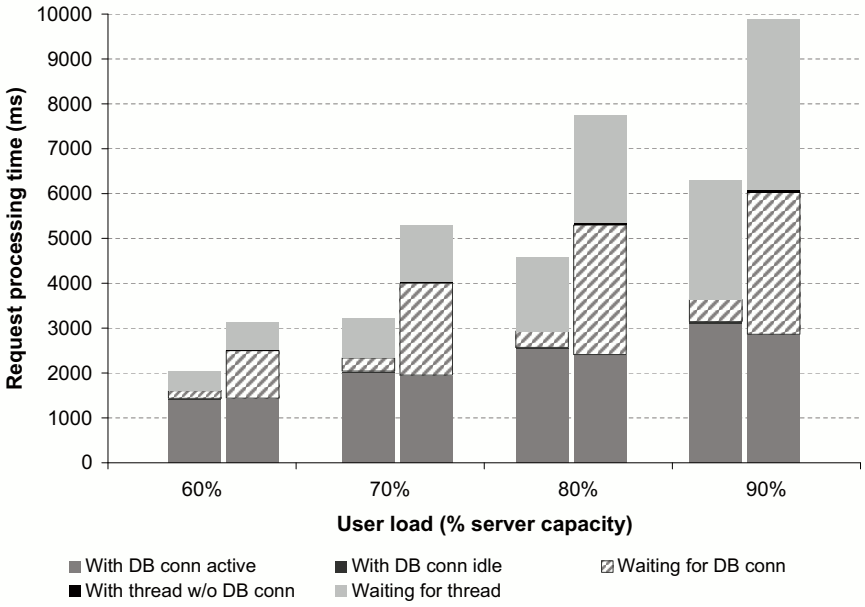


Figure 3. Breakdown of request processing time for the Search request (left column: request-wide database connection caching, right column: transaction-wide caching).

caching mechanism. We distinguish among the following phases of request execution. **Waiting for thread** is the time spent waiting for a thread to process the request. **With thread w/o DB connection** is the time spent processing the request without holding a database connection. **Waiting for DB connection** is the time spent waiting for a database connection, i.e. while being blocked on the `Datasource.getConnection()` call (the call to `Datasource.getConnection()` requests a database connection from the pool, if one is not cached by the request). **With DB connection active** is the time spent working with the database connection, i.e. between the return of the call `Datasource.getConnection()` and the call to `Connection.close()` (the call to `Connection.close()` is actually performed on a wrapper object (Section 2.1), so the connection is either kept cached for the request or returned to the pool, depending on which connection caching policy — our request-wide or default transaction-wide — is used). **With DB connection idle** is the time spent processing the request with an idle database connection cached, that is, when the connection is cached for the request between the call to `Connection.close()` and the next call to `Datasource.getConnection()` or the end of request execution. Note, that we don't track database activities performed over database connections and the periods of "working with a database connection" are demarcated by the calls to `Datasource.getConnection()` (start) and `Connection.close()` (end). Note also that in the case of the request-wide database connection caching, **waiting for thread** corresponds to w_i^{THR}, **with thread w/o DB connection** — to p_i, **waiting for DB connection** — to w_i^{DB}, and **with DB connection active** and **with DB connection idle**, combined, constitute q_i.

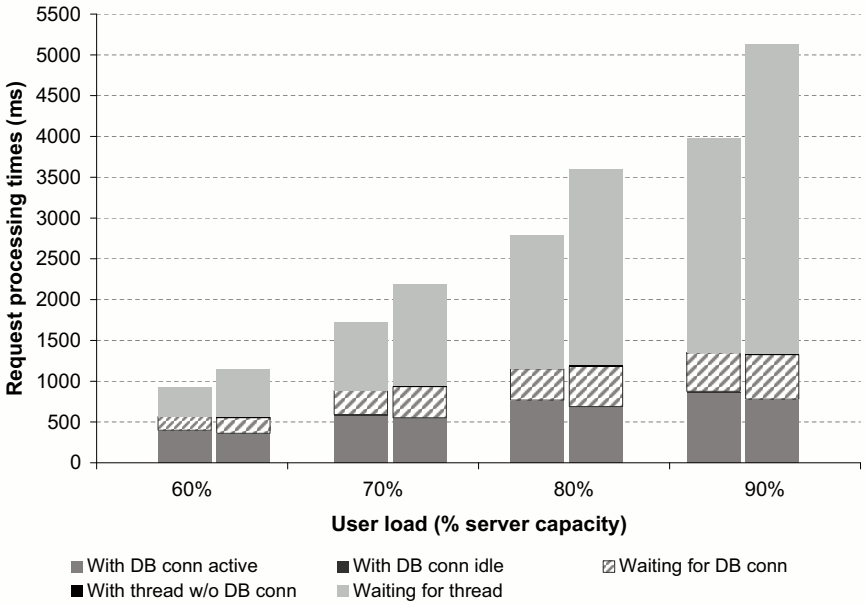


Figure 4. Breakdown of request processing time for the Buy Request request (left col.: request-wide database connection caching, right col.: transaction-wide caching).

To better understand the relative performance of the two connection caching methods, it is important to know how many times a database connection is requested from the pool, for different request types. During the execution of the Search request a database connection is requested from the pool 101 times (one time for the SQL query that returns IDs of certain 50 items, and 2 times for an EJB method invocation on each of the 50 entity EJB application components, see explanation in Section 3). Buy Request results in 10 requests for a database connection, and Buy Confirm incurs 1 request for a database connection, because this service request is transactional, and the obtained database connection is cached even in the default transaction-wide database connection caching mechanism.

The first thing to notice is that **with DB connection idle** times are very small and are negligible compared with **with DB connection active** times (they aren't even seen on some charts). This means that the request-wide database connection caching does not waste much database connection resources. Individual times of waiting for database connection are larger in the request-wide connection caching (than in the transaction-wide connection caching), but this effect is compensated by the fact that a service request has only to endure one such waiting time period. Request-wide connection caching is beneficial for service requests that obtain (and return) database connection from the pool many times — compare **waiting for DB connection** times for the Search and Buy Confirm service requests, which obtain database connection from the pool 101 times and 1 time, respectively. The time to process a request after it was assigned to a thread, for the Search request, is significantly larger in the default transaction-wide connection caching, because of the aforementioned effect. Given that Search requests constitute a large portion of user load that we used (namely, 32%), the average time requests (of all types) spend with a thread assigned is larger (in the default transaction-wide connection

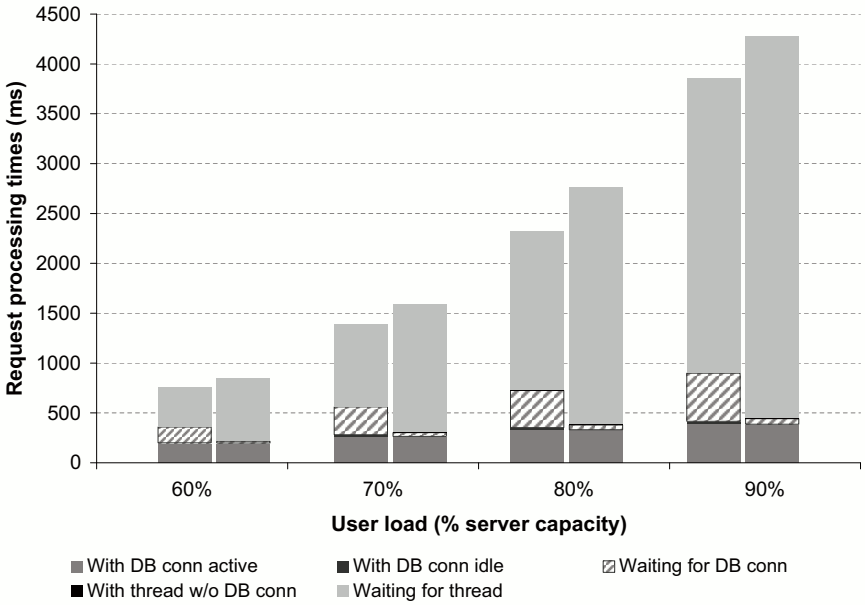


Figure 5. Breakdown of request processing time for the Buy Confirm request (left col.: request-wide database connection caching, right col.: transaction-wide caching).

caching) and, as a consequence, **waiting for thread** times are larger for *all* request types. This results in larger overall request processing times for *all* request types, in the transaction-wide connection caching, compared with the proposed request-wide database connection caching.

The main conclusion that we can draw by analyzing the relative performance of both database connection caching methods, is that our proposed request-wide database connection caching is beneficial in situations where a large portion of service requests is comprised of non-transactional requests implemented in a way that database connections are requested from the pool many times per single service request execution. This characteristic is typical of the applications and workloads we target in this work.

5.3. Evaluation of the method for computing the optimal resource pool sizes

In this Section, we evaluate the proposed method for computing the optimal number of server threads and database connections (Section 4) on the TPC-W application, which is configured as described in Section 5.1.

In step 1, the values of p_i and q_i are gathered through the request execution profiling support using the user load parameters shown in Table II ("Profiling load" column). We use the following data points: all pairs (M, N), where $M = 5, 10, 15, 20, 30, 50$, $N = 1, 2, 4, 5, 10, 15, 20, 30, 40$, and $M \geq N$. Reasonable application performance was achieved with $5 \leq M \leq 30$, so we concentrate on this region.

Table II. Parameters of user loads used in the evaluation experiments: breakdown of load by request types (V_i/L) and average session length (L).

Request type	"Profiling load" (step 1)	"Load 1"	"Load 2"
Home	6.82%	5.53%	5.37%
Search	**31.93%**	**30.31%**	**48.07%**
Item	**42.73%**	**57.73%**	**41.55%**
Add To Cart	4.76%	1.76%	1.39%
Cart	2.03%	0.7%	0.53%
Register	3.58%	1.32%	1.03%
Buy Request	3.58%	1.32%	1.03%
Buy Confirm	3.58%	1.32%	1.03%
Total	100%	100%	100%
Average session length L (number of requests)	14.67	18.07	18.62

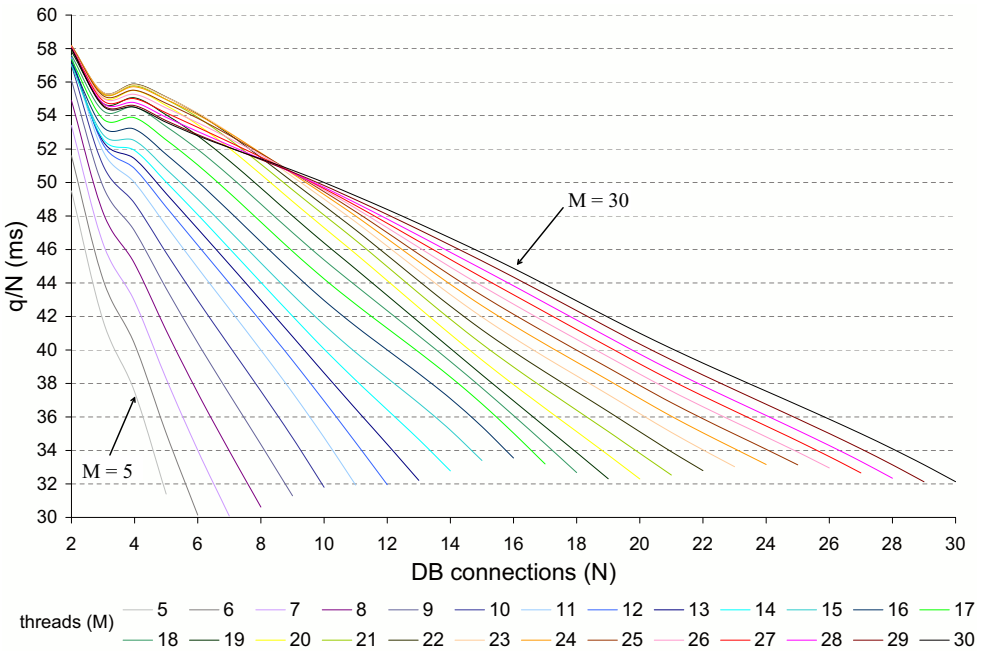


Figure 6. q_i/N for the Search request.

In step 2, we interpolate functions $p_i(M,N)$ and $q_i(M,N)$ for the region $5 \leq M \leq 30$, $0 \leq N \leq M$. Figs. 6, 7, and 8 show examples of interpolated functions — $q_i(M,N)/N$ for the Search and Item requests, and function $p_i(M,N)$ for the Home request, respectively. Each curve on these charts corresponds to the function with fixed M (number of threads), $M = 5,\ldots,30$, and is parameterized by N (number of database connections).

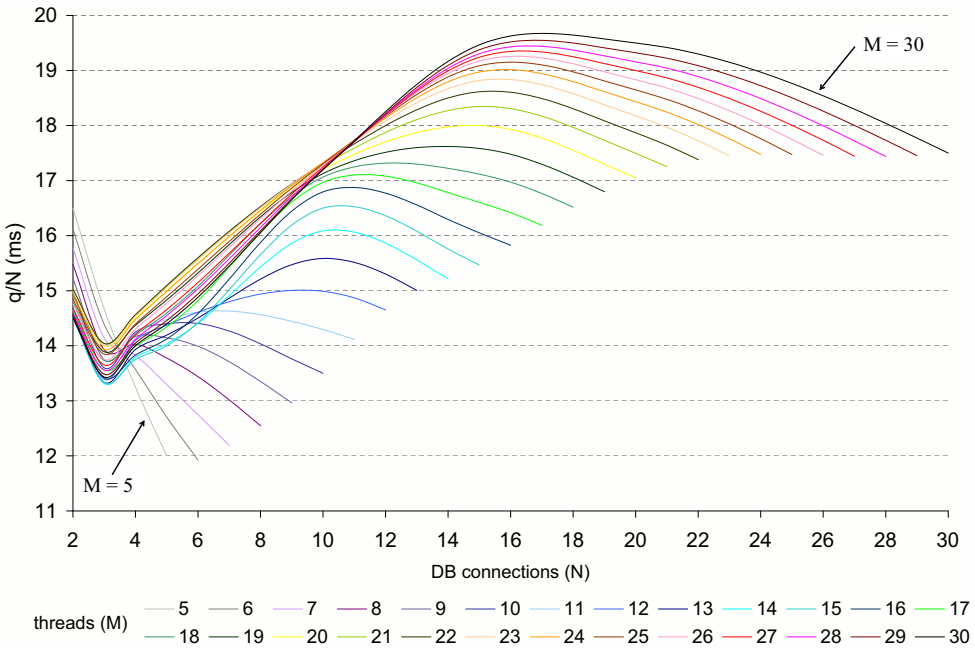


Figure 7. q_i/N for the Item request.

The shapes of the function curves may be quite complex. They reflect the behavior of different request types, relative to the sizes of the thread and database connection pools, and show how requests scale with increased number of threads and database connections. Obviously, functions $p_i(M, N)$ and $q_i(M, N)$ depend on hardware, middleware, and database configurations. But most notably, we believe, they depend on the functionality and the implementation of the service requests, i.e., on the nature of the underlying work being done by the requests, on the way low-level resources (such as CPU, memory, IO) are used, and on the way concurrent requests interfere with each other (i.e., database locking and data contention issues). For example, the Search request, which performs complex read-only database queries, scales well and enjoys better performance with increased degree of parallelism (Fig 6), while the Item request shows the best performance with relatively small number of threads and database connections ($M, N = 5, \ldots, 8$, see Fig. 7).

In step 3 we get the values of maximum sustainable session throughput $\lambda(M, N)$, for every combination of M and N. We compute these values for the same load as we used in the "profiling tests" (Table II). Fig. 9 shows function $\lambda(M, N)$, computed by our method.

We define $\lambda(N)$ as the maximum sustainable session throughput as a function of N, where M is chosen to achieve the best throughput for a given N:

$$\lambda(N) = \lambda(M_0, N) \mid M_0 = \underset{M}{\operatorname{argmax}}\, \lambda(M, N) \tag{5}$$

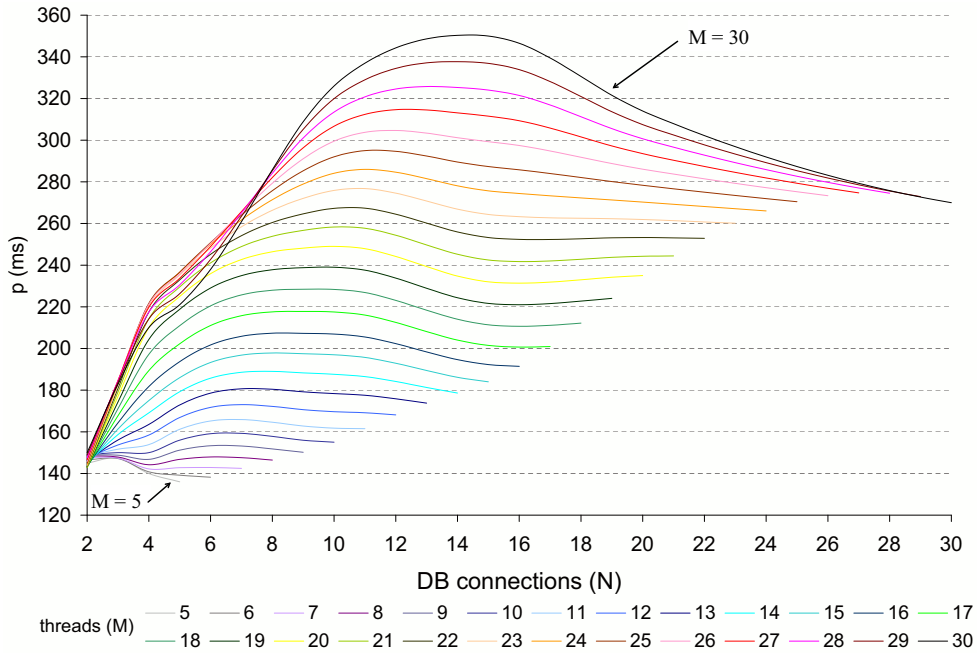


Figure 8. p_i for the Home request.

Fig. 9 also shows $\lambda(N)$ computed by our method ("$\lambda(N)$ meth"), and obtained experimentally ("$\lambda(N)$ exp"). To obtain the latter, for each optimal pair of (M_0, N) computed by the method (M_0 defined by equation (5)) we run tests with ($M_0 - 1$, N), (M_0, N), ($M_0 + 1$, N), and ($M_0 - 2$, N), ($M_0 + 2$, N) if needed, to determine the maximum session throughput in each case and confirm that M_0 is indeed the optimal for a given N.

As a matter of fact, it is difficult to tell the *exact* throughput of a server configuration due to non-determinism in its behavior, especially for user loads that we use (Poisson session arrivals, random inter-request times, see Section 5.1). One can tell only *approximate* request (session) throughput, which can be defined, for example, as the maximum λ, at which 60% of test runs complete with a request success rate of 100%. Therefore, the values of λ (session arrival rate) used in the tests determining the actual maximum session throughout are incremented with a granularity of 0.05 (e.g., 2.65, 2.70, 2.75, ...).

Our method succeeds in determining the value of M that achieves the highest throughput for a given N (also referred to as "the optimal M for a given N"), and vice versa. However, it might not be exactly precise in determining the value of actual maximum session throughput, which may well happen to be a little lower than projected by the method. This effect can be attributed to the burstiness of the incoming user requests and to the thread and database connection "context switching": a thread that releases a database connection notifies the next waiting thread, and it may take some CPU cycles before the waiting thread grabs the released database connection; in other words, a database connection occurs to be occupied for a slightly greater time, than can be recorded. However, the actual session throughput

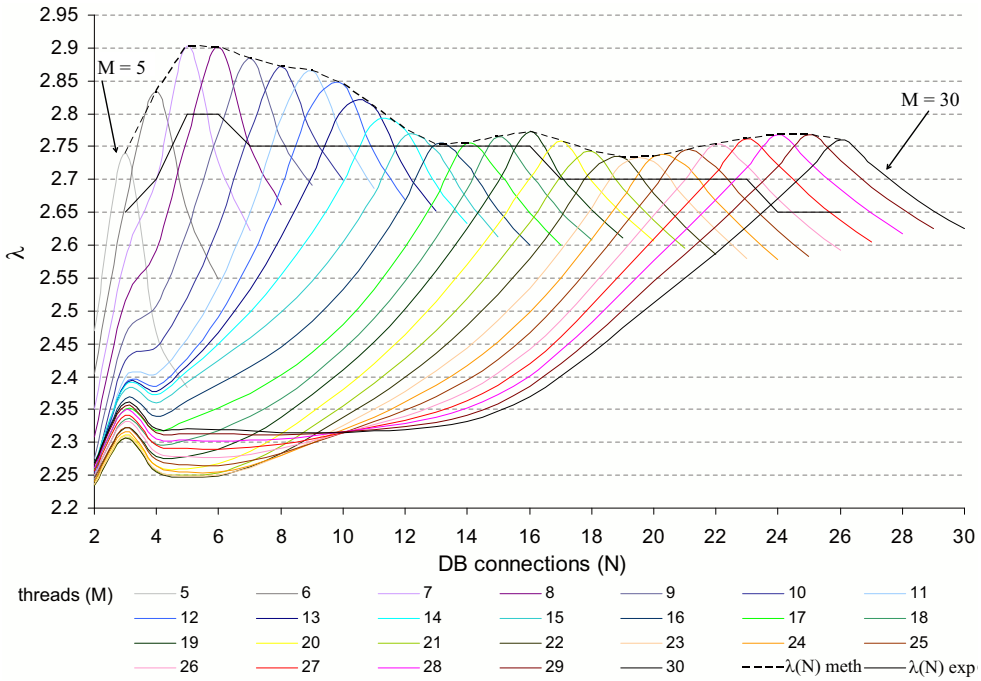


Figure 9. $\lambda(M,N)$ and $\lambda(N)$, computed by our method ("$\lambda(N)$ meth") and obtained experimentally ("$\lambda(N)$ exp"), for the "profiling load."

(determined by the experiments) always lies within a 5% error margin of the value predicted by the method.

It is interesting to notice that the optimal pairs of (M,N) are ones where N is very close to M, e.g., (8,6), (14,11), (30,26). This means that only a few additional threads are needed to do processing of requests that do not access the database. This is also seen in the fact that under load the values of q_i's increase much higher than do the values of p_i's (Table I). This indicates that requests, after being assigned to a server thread, spend a dominant portion of their processing time working with the database connection.

In the next series of experiments we tested the ability of our method to work with user loads that differ from the one used in the "profiling tests." In preliminary tests we noticed that the dependence of p_i's and q_i's on the incoming request mix (V_i) is very weak, i.e., their values change insignificantly when the request load parameters V_i stay *close enough* to their initial values. This suggests that it might be possible to gather the values of p_i's and q_i's for some average (*representative*) user load, and later use them to compute the optimal number of threads and database connections for loads that differ from the one used in the "profiling tests."

To verify this, we test our method on two user loads that differ from the "profiling load", but stay relatively *close* to it (with the values of V_i varying by no more than $\pm$15-20%). In computations, we use the same p_i's and q_i's obtained for the "profiling load". The parameters of these loads ("Load 1" and "Load 2") are shown in Table II. As the "profiling load," these two loads are chosen to be representative

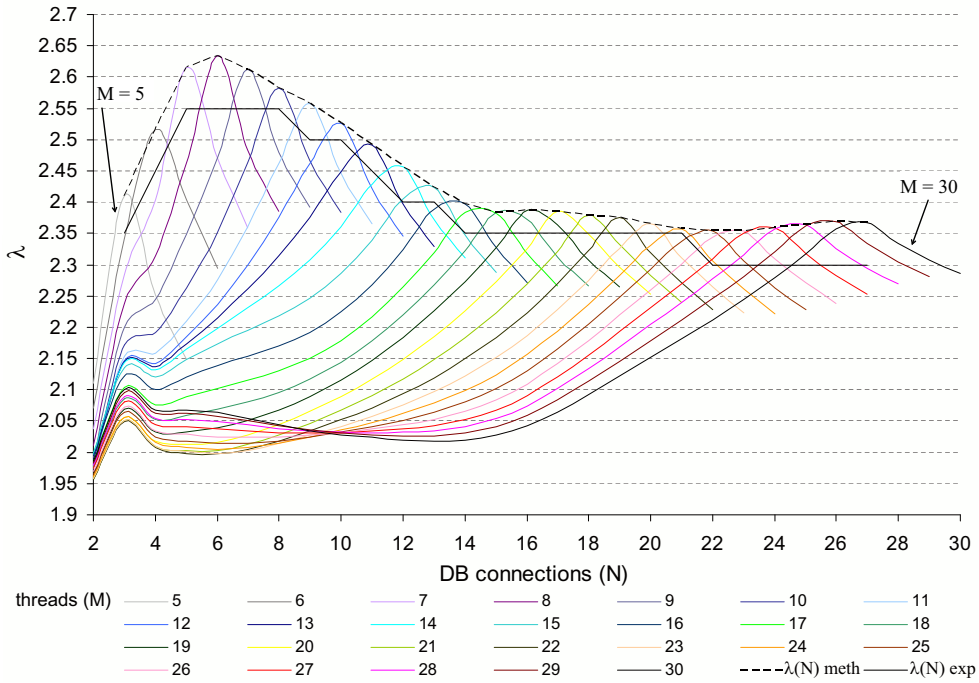


Figure 10. $\lambda(M,N)$ and $\lambda(N)$, computed by our method ("$\lambda(N)$ meth") and obtained experimentally ("$\lambda(N)$ exp"), for the "Load 1" user load.

of real-life shopping scenarios, where Search and Item requests dominate all other request types. The chosen loads represent the two extremes of the load spectrum — "Load 1" has a dominant portion of Item requests, while "Load 2" has a greater number of Search requests.

As with the "profiling load" previously in the paper, we use our method to compute the values of $\lambda(M,N)$ and $\lambda(N)$, and perform a series of test runs to obtain the values of $\lambda(N)$ experimentally. Figs. 10 and 11 show the results of these computations and experiments for "Load 1" and "Load 2," respectively. As we see, the method works well for both user loads. The method is able to compute the optimal number of threads and database connections and the actual values of session throughput lie within a 5% margin of the values predicted by the method.

Note that if the actual user load deviates considerably from the "profiling load", then it may not be possible anymore to use the old values of $p_i(M,N)$ and $q_i(M,N)$. In such situation, the profiling tests should be rerun with the newer user load and new values of $p_i(M,N)$ and $q_i(M,N)$ should be obtained. One of the interesting directions for future work is to explore the question of when should a load be deemed *close enough* to the original load, and when should it be deemed *a load that deviated significantly*. It would be also beneficial to define certain management policies that would trigger alerts indicating that the newer values of $p_i(M,N)$ and $q_i(M,N)$ should be obtained.

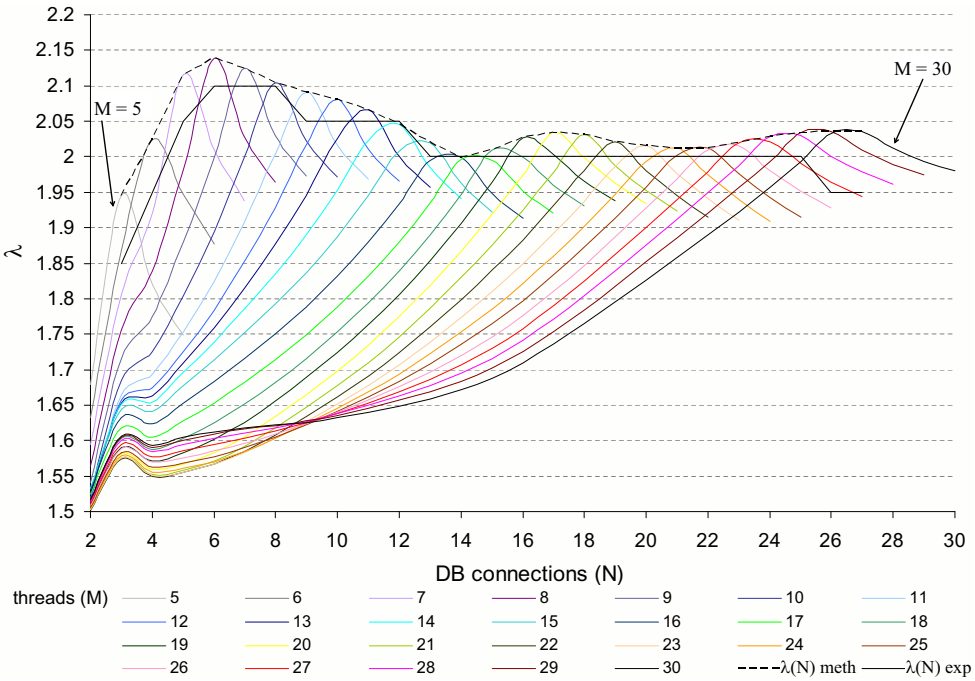


Figure 11. $\lambda(M, N)$ and $\lambda(N)$, computed by our method ("$\lambda(N)$ meth") and obtained experimentally ("$\lambda(N)$ exp"), for the "Load 2" user load.

6. Related Work and Discussion

There is a vast body of research work in the area of analytical modeling of Internet applications and underlying server environments, with different interconnected goals pursued: (1) *capacity provisioning* allows one to determine how much capacity is needed for an Internet application to allow it to service the specified workload [24, 25, 26]; (2) *performance prediction* enables the response time of the application to be determined for a given workload and a given hardware and software configuration [27, 5, 28]; (3) *application configuration* enables various configuration parameters of the application to be determined for a certain performance goal [4]; and (4) *bottleneck identification and tuning* enables system bottlenecks to be identified for purposes of performance tuning [27]. Our study falls into the last-named category.

The vast majority of analytical models of Internet applications utilize the apparatus of Queueing Theory [29], while some use Markov chains [30] and control-theoretical approaches [31, 32]. Modeling of single-tier Internet applications, of which HTTP servers are the most common example, was studied extensively [30, 33, 34]. Most of these studies also assume static web content. Several efforts have focused on the modeling of multi-tier applications [35, 27, 26]. However, many of these efforts either make simplifying assumptions or are based on simple extensions of single-tier models. A number of papers have taken the approach of modeling only the most constrained or the most bottlenecked tier of the application. For instance, [24] considers the problem of provisioning servers for only the middle

(EJB) tier. Other efforts have modeled the entire multi-tier application using a single queue [5]. While these efforts have made good progress, analytical modeling of Internet applications remains a hard problem, because of complex structural and behavioral properties of these applications.

Acknowledging the above difficulties in modeling modern Internet applications, we have taken a different avenue in this work. Instead of trying to come up with a *precise* (e.g., queuing) model of an Internet application, we propose a rather simplified approach, where the critical system configuration parameters (thread and database connection pool sizes) are computed *indirectly*, based on some observations performed in a set of offline profiling tests. The result is a practical method that allows to accurately compute the size of critical resource pools, without the need to build overly complex and sometimes intractable, but precise analytical model of the Internet application. Of course, there is a price to pay for such simplicity: one has to run a set of aforementioned offline profiling tests. However, we believe that this is a small drawback, because real-world (commercial) Internet services are anyway subjected to some testing and profiling, before being released as a commercial offering.

The (intentional) simplicity of our approach is also manifested in the fact that in our model we do not use information about various wait times in the system: w^{THR} and w^{DB} (1), which may seen somewhat counterintuitive at a first glance. The reason for this is that in order to compute maximum sustainable session throughput $\lambda(M, N)$, we consider an "idealistic" situation where requests do not wait for resources to become available and all threads and database connections are always busy processing requests. In this setting, it is the values of p_i and q_i that determine the value of $\lambda(M, N)$. This approximation, as we have shown, works well for regular (non-bursty) user load (indeed, we model the flow of incoming new user sessions as a Poisson process), but may potentially be inaccurate for extreme bursty user loads. One interesting direction for future work is to see how the proposed model behaves with bursty user loads, and to modify the model to take into account the burstiness parameters of that load, if the model does not produce adequate results. It may be the case that we will have to take into account various wait times in the system, e.g., the values of w^{THR} and w^{DB}, to come up with a more accurate model for bursty user loads, and almost certainly the resulting model will be far more complex and far less tractable than the one presented in this paper.

In our method of computing the optimal sizes of critical server resource pools, we make functions $p_i(M, N)$ and $q_i(M, N)$ operate over a domain that covers all values of the number of threads and database connections (M and N) that potentially achieve reasonable application performance. For all database-centric web applications that we experimented with (including the TPC-W application) this was the region of $0 < N \leq M \leq 30$. In step 1 of the method, we perform profiling tests for a limited data point subset of the functions' domain (roughly every 5-th value of M and N). This data point set does not seem overly large if the profiling tests (step 1) are run only once. But if this step needs to be run more often (e.g., if the user load deviates significantly from the "training" load used in step 1), there may be a need to optimize the profiling step, e.g., by reducing the data point set, and therefore minimizing the size of the profiling tests. In addition to that, the adjusted data point set may need to be more dense (or more sparse) than the one we chose (i.e., roughly every 5-th value of M and N), to interpolate the functions better. It is a very interesting direction for future work to see if the choice of an optimal *interpolation region* and the *data point set* on which the profiling tests are run, can be automated in any way.

The proposed method to compute the optimal sizes of critical server resource pools has one additional advantage compared to other contemporary methods for analytical modeling of Internet services. Unlike some other models [5, 27], the method *does not require the knowledge of application structure*. This is the case because the method works at the level of *physical server resource tiers*, rather than at the level of *logical application tiers* [27], thus being oblivious to the actual logical (e.g., component) structure of the Internet application.

Note also that the method proposed in this paper is orthogonal to other methods of improving performance of Internet services, such as request admission control and scheduling [19, 20, 21, 7], service differentiation [31, 36], and service distribution [37, 38]. In this paper, we present our techniques in a centralized setting, to focus on the essence and benefits of the proposed method to compute the optimal size of exclusively-held resource pools (e.g., threads and database connections). We believe that the method will show its utility in a distributed setting as well, where it can be independently applied at every web/application/database server that sees concurrent requests competing for server resources that are held exclusively by the executing requests.

7. Conclusion

In this paper, we looked at performance of Internet services implemented as applications hosted on web application (middleware) servers. First, we presented a model for service request execution with 2-tier exclusive resource holding (1st tier: threads, 2nd tier: database connections). In this model, we advocated the use of a database connection pooling mechanism that caches database connections for the duration of the service request execution (request-wide database connection caching).

Second, we presented a methodology that computes the optimal size of critical server resource pools (threads and database connections) in a web application server, for a given Internet application, its server and database environment. The methodology is built on the above model for request execution with 2-tier exclusive resource holding. The method uses information about fine-grained server resource utilization by service requests of different types, obtained through request execution profiling in a limited set of offline ("profiling") tests, where the actual server environment is subjected to an artificial user load. Under real operating conditions, the method takes as input the information about the actual incoming request flow, obtained through online request profiling, and computes the optimal number of threads and database connections that achieve the highest request throughput, for a given mix of incoming user requests, thus enabling the possibility of dynamic adaptation of web application server environment to changing user load conditions.

We evaluated the proposed methodologies by testing them on the TPC-W application in a web application server environment, augmented with a request profiling infrastructure. We showed the performance superiority of the proposed request-wide database connection caching, compared to standard transaction-wide database connection caching, for the class of heavy database-centric "read-mostly" applications, such as TPC-W web transactional benchmark. Our results also show that our method to compute the optimal size of critical server resource pools is always able to accurately compute the optimal number of threads and database connections, and the value of maximum sustainable request throughput computed by the method always lies within a 5% margin of the actual value determined experimentally.

REFERENCES

1. IBM Corporation. WebSphere Platform. `http://www.ibm.com/websphere` [16 January 2010].
2. Oracle WebLogic Application Server. `http://www.oracle.com/weblogicserver` [16 January 2010].
3. Red Hat JBoss Java Application Server. `http://www.jboss.org` [16 January 2010].
4. Chung I, Hollingsworth J. Automated cluster-based web service performance tuning. *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC04)*, Honolulu, HI, U.S.A., 2004.
5. Stewart C, Shen K. Performance modeling and system management for multi-component online services. *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*, Boston, MA, U.S.A., 2005.

6. Pacifici G, Spreitzer M, Tantawi A, Youssef A. Performance management for cluster-based web services. *IEEE Journal on Selected Areas in Communications* 2005; **23**(12):2333–2343.
7. Totok A, Karamcheti V. RDRP: Reward-driven request prioritization for e-Commerce web sites. *Electronic Commerce Research and Applications* 2010; **9**(6):549–561.
8. Sun Microsystems. Java Platform Enterprise Edition (Java EE). `http://java.sun.com/javaee/` [16 January 2010].
9. Transaction Processing Performance Council. TPC-W: Transactional Web e-Commerce Benchmark. `http://www.tpc.org/tpcw/` [16 January 2010].
10. Sun Microsystems. Java Database Connectivity (JDBC) technology. `http://java.sun.com/products/jdbc/` [16 January 2010].
11. New York University. TPC-W-NYU: A Java EE implementation of the TPC-W benchmark. `http://www.cs.nyu.edu/totok/professional/software/tpcw/tpcw.html` [16 January 2010].
12. Marinescu F. *EJB Design Patterns*. John Wiley & Sons, 2002.
13. Sun Microsystems. Java Servlet Technology. `http://java.sun.com/products/servlet/` [16 January 2010].
14. Sun Microsystems. Enterprise JavaBeans (EJB) technology. `http://java.sun.com/products/ejb/` [16 January 2010].
15. Kincaid D, Cheney E. *Numerical Analysis: Mathematics of Scientific Computing*. Brooks Cole: Stamford, CT, U.S.A., 2001.
16. Jetty HTTP Server and Servlet Container. `http://jetty.mortbay.org` [16 January 2010].
17. Sun Microsystems. MySQL Database. `http://www.mysql.com/` [16 January 2010].
18. Totok A, Karamcheti V. Modeling of concurrent web sessions with bounded inconsistency in shared data. *Journal of Parallel and Distributed Computing* 2007; **67**(7):830–847.
19. Chen H, Mohapatra P. Session-based overload control in QoS-aware web servers. *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'02)*, New York, NY, U.S.A., 2002.
20. Elnikety S, Nahum E, Tracey J, Zwaenepoel W. A method for transparent admission control and request scheduling in dynamic e-Commerce web sites. *Proceedings of the International World Wide Web Conference (WWW'04)*, New York, NY, U.S.A., 2004.
21. Singhmar N, Mathur V, Apte V, Manjunath D. A combined LIFO-priority scheme for overload control of e-Commerce web servers. *Proceedings of the IEEE RTSS International Infrastructure Survivability Workshop (IISW'04)*, Lisbon, Portugal, 2004.
22. Menascé D, Almeida V, Fonseca R, Mendes M. A methodology for workload characterization of e-Commerce sites. *Proceedings of the 1st ACM Conference on Electronic Commerce (EC'99)*, Denver, CO, U.S.A., 1999.
23. Menascé D, Almeida V, Riedi R, Ribeiro F, Fonseca R, W Meira Jr. In search of invariants for e-Business workloads. *Proceedings of the 2nd ACM Conference on Electronic Commerce (EC'2000)*, Minneapolis, MN, U.S.A., 2000.
24. Villela D, Pradhan P, Rubenstein D. Provisioning servers in the application tier for e-Commerce systems. *Proceedings of the 12th IEEE International Workshop on Quality of Service (IWQoS'04)*, Montreal, Canada, 2004.
25. Zhang Q, Cherkasova L, Mathews G, Greene W, Smirni E. A capacity planning framework for multi-tier enterprise services with real workloads. *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management (IM'07)*, Munich, Germany, 2007.
26. Zhang Q, Cherkasova L, Mi N, Smirni E. A regression-based analytic model for capacity planning of multi-tier applications. *Journal on Cluster Computing. Special Issue on Autonomic Computing* 2008; **11**(3):197–211.
27. Urgaonkar B, Pacifici G, Shenoy P, Spreitzer M, Tantawi A. An analytical model for multi-tier Internet services and its applications. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Banff, Alberta, Canada, 2005.
28. Kelly T, Zhang A. Predicting performance in distributed enterprise applications. HP Laboratories Technical Report HPL-2006-76, May 2006.
29. Kleinrock L. *Queueing Systems*. Wiley: New York, 1975.
30. Menascé D. Web server software architectures. *IEEE Internet Computing* 2003; **7**(6):78–81.
31. Lu C, Abdelzaher T, Stankovic J, Son S. A feedback control approach for guaranteeing relative delays in web servers. *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, Taipei, Taiwan, 2001.
32. Abdelzaher TF, Shin KG, Bhatti N. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems* 2002; **13**(1):80–96.
33. Doyle R, Chase J, Asad O, Jin W, Vahdat A. Model-based resource provisioning in a web service utility. *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Seattle, WA, U.S.A., 2003.
34. Urgaonkar B, Shenoy P. Cataclysm: Handling extreme overloads in Internet services. *Proceedings of the 23rd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'04)*, St. John's, Newfoundland, Canada, 2004.
35. Kamra A, Misra V, Nahum E. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. *Proceedings of the 12th IEEE International Workshop on Quality of Service (IWQoS'04)*, Montreal, Canada, 2004.
36. Chen H, Iyengar A. A tiered system for serving differentiated content. *World Wide Web Journal* 2003; **6**(4):331–352.
37. Gao L, Dahlin M, Nayate A, Zheng J, Iyengar A. Application specific data replication for edge services. *Proceedings of the International World Wide Web Conference (WWW'03)*, Budapest, Hungary, 2003.

38. Llambiri D, Totok A, Karamcheti V. Efficiently distributing component-based applications across wide-area environments. *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, Providence, RI, U.S.A., 2003.