J2EE Performance and Scalability - From Measuring to Predicting

Samuel Kounev, *Member, IEEE*

***Abstract*— J2EE applications are becoming increasingly ubiquitous and with their increasing adoption, performance and scalability issues are gaining in importance. For a J2EE application to perform well and be scalable, both the platform on which it is built and the application design must be efficient and scalable. Industry-standard benchmarks such as the SPECjAppServer set of benchmarks help to evaluate the performance and scalability of alternative platforms for J2EE applications, however, they cannot be used to evaluate the performance and scalability of concrete applications built on the selected platforms. In this paper, we present a systematic approach for evaluating and predicting the performance and scalability of J2EE applications based on modeling and simulation. The approach helps to identify and eliminate bottlenecks in the application design and ensure that systems are designed and sized to meet their quality of service requirements. We introduce our approach by showing how it can be applied to the SPECjAppServer2004 benchmark which is used as a representative J2EE application. A detailed model of a SPECjAppServer2004 deployment is built in a step-by-step fashion and then used to predict the behavior of the system under load. The approach is validated by comparing model predictions against measurements on the real system.**

***Index Terms*— Performance modeling and prediction, software verification, performance evaluation, distributed systems**

I. Introduction

The Java 2 Enterprise Edition (J2EE) platform is becoming increasingly ubiquitous as enabling technology for modern enterprise applications. The aim of J2EE is to make it easier to build scalable, reliable and secure applications by leveraging middleware services and commercial-off-the-shelf (COTS) components provided by the industry. However, practice has proven that developing well-performing and scalable J2EE applications is not as easy as it sounds. Building on a scalable platform using well-tested and proven components does not automatically provide for good performance and scalability. A given set of components may perform well in one combination and rather bad in another. In fact, by hiding the complexity of the innerworkings of components and the underlying middleware infrastructure, J2EE makes developers much more dependent on the latter and makes it easy to introduce processing inefficiencies and system bottlenecks by inadvertence. To avoid the pitfalls of inadequate *Quality of Service (QoS)*, it is important to analyze the expected performance characteristics of systems during all phases of their life cycle. However, as systems grow in size and complexity, analyzing their performance becomes a more and more challenging task. System architects and deployers are often faced with the following questions:

1) Which platform (hardware and software) would provide the best cost/performance ratio for a given application?[1]
2) How do we ensure that the selected platform does not have any inherent scalability bottlenecks?
3) What performance would a deployment of the application on the selected platform exhibit under the expected workload and how much hardware would be needed to meet the Service Level Agreements (SLAs)?
4) How do we ensure that the application design does not have any inherent scalability bottlenecks?

Answering the first two questions requires *measuring* the performance and scalability of alternative hardware and software platforms which is typically done using benchmarks. Answering the second two questions requires *predicting* the performance of a given application deployed on a selected platform which is normally done through load testing or performance modeling.

A number of benchmarks have been developed in the past decade that can be used to measure the performance and scalability of J2EE platforms (see for example [1]–[8]). Benchmarks help to compare platforms and validate them, however, they can also be exploited to study the effect of different platform configuration settings and tuning parameters on the overall system performance [9]–[11]. Thus, benchmarking helps to select a platform for a given application and configure (tune) the platform for optimal performance. However, while building on a scalable and optimized platform is a necessary condition for achieving good performance and scalability, unfortunately, it is not sufficient. The application built on the selected platform must also be designed to be efficient and scalable. This takes us to the second problem mentioned above. How do we predict the performance and scalability of J2EE applications in order to ensure that there are no design bottlenecks and that enough resources are provided to guarantee adequate performance?

In this paper, we present a case study with the industry-standard SPECjAppServer2004 [2] benchmark in which the latter is used as a representative J2EE application in order to introduce a systematic approach for predicting the perfor-

The author is with the Department of Computer Science, Darmstadt University of Technology, 64289 Darmstadt, Germany. E-mail: skounev@acm.org, skounev@informatik.tu-darmstadt.de or skounev@spec.org

This work was partially funded by the German Research Foundation (DFG) as part of the PhD program ”Enabling Technologies for E-Commerce” at Darmstadt University of Technology.

[1] In this paper, we use the terms ”system” and ”application” interchangeably.

[2] SPECjAppServer is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjAppServer2004 results or findings in this publication have not been reviewed or accepted by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result. The official web site for SPECjAppServer2004 is located at http://www.spec.org/osg/jAppServer2004.

mance and scalability of J2EE applications. The approach used is based on performance modeling and simulation. A detailed model of a SPECjAppServer2004 deployment is built in a step-by-step fashion. The model is validated and used to predict the system performance for several deployment configurations and workload scenarios of interest. In each case, the model is analyzed by means of simulation. In order to validate the approach, the model predictions are compared against measurements on the real system. It is demonstrated how some complex aspects of system behavior such as composite transactions and asynchronous processing can be modeled.

The rest of the paper is organized as follows. We start with an introduction of the SPECjAppServer2004 benchmark in Section II. In Section III, a detailed model of a SPECjAppServer2004 deployment is built in a step-by-step fashion. The model is validated and used to predict the performance of the system for several different scenarios. It is shown how the model predictions can be used for performance analysis and capacity planning. Finally, in Section IV, the paper is wrapped up with some concluding remarks.

II. The SPECjAppServer2004 Benchmark

SPECjAppServer2004 is a new industry standard benchmark for measuring the performance and scalability of J2EE application servers. SPECjAppServer2004 was developed by SPEC's Java subcommittee which includes BEA, Borland, Darmstadt University of Technology, Hewlett-Packard, IBM, Intel, Oracle, Pramati, Sun Microsystems and Sybase. It is important to note that even though some parts of SPECjAppServer2004 look similar to SPECjAppServer2002, SPECjAppServer2004 is much more complex and substantially different from previous versions of SPECjAppServer. It implements a new enhanced workload that exercises all major services of the J2EE platform in a complete end-to-end application scenario. The SPECjAppServer2004 workload is based on a distributed application claimed to be large enough and complex enough to represent a real-world e-business system [2]. It has been specifically modeled after an automobile manufacturer whose main customers are automobile dealers. Dealers use a Web based user interface to browse the automobile catalogue, purchase automobiles, sell automobiles and track dealership inventory.

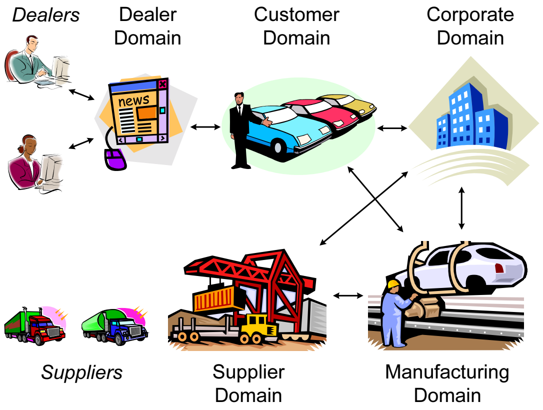


Fig. 1. SPECjAppServer2004 business model.

As depicted on Figure 1, SPECjAppServer2004's business model comprises five domains: customer domain dealing with customer orders and interactions, dealer domain offering Web based interface to the services in the customer domain, manufacturing domain performing ”just in time” manufacturing operations, supplier domain handling interactions with external suppliers, and corporate domain managing all customer, product and supplier information. The customer domain hosts an order entry application that provides some typical online ordering functionality. Orders for more than 100 automobiles are called *large orders*. The dealer domain hosts a Web application (called dealer application) that provides a Web based interface to the services in the customer domain. Finally, the manufacturing domain hosts a manufacturing application that models the activity of production lines in an automobile manufacturing plant.

There are two types of production lines, *planned lines* and *large order lines*. Planned lines run on schedule and produce a predefined number of automobiles. Large order lines run only when a large order is received in the customer domain. The unit of work in the manufacturing domain is a *work order*. Each work order moves along three virtual stations which represent distinct operations in the manufacturing flow. In order to simulate activity at the stations, the manufacturing application waits for a designated time (333 ms) at each station. Once the work order is complete, it is marked as completed and inventory is updated. When inventory of parts gets depleted suppliers need to be located and *purchase orders (POs)* need to be sent out. This is done by contacting the supplier domain which is responsible for interactions with external suppliers.

All the activities and processes in the five domains described above are implemented using J2EE components (Enterprise Java Beans, Servlets and Java Server Pages) assembled into a single J2EE application that is deployed in an application server running on the *System Under Test (SUT)*. The only exception is for the interactions with suppliers which are implemented using a separate Java servlet application called *supplier emulator*. The latter is deployed in a Java-enabled Web server on a dedicated machine. The workload generator is implemented using a multi-threaded Java application called *SPECjAppServer driver*. The driver is made of two components - *manufacturing driver* and *DealerEntry driver*. The manufacturing driver drives the production lines in the manufacturing domain and exercises the manufacturing application. It communicates with the SUT through the RMI (Remote Method Invocation) interface. The DealerEntry driver emulates automobile dealers using the dealer application in the dealer domain to access the services of the order entry application in the customer domain. It communicates with the SUT through HTTP and exercises the dealer and order entry applications using three operations referred to as *business transactions*:

1) Browse - browses through the vehicle catalogue
2) Purchase - places orders for new vehicles
3) Manage - manages the customer inventory

Each business transaction emulates a specific type of client session comprising multiple round-trips to the server. For example, the Browse transaction navigates to the vehicle catalogue web page and then pages a total of thirteen times, ten forward and three backwards. A relational DBMS is used for data persistence and all data access operations use entity beans which are mapped to tables in the SPECjAppServer database. Data access components follow the guidelines in [12] to provide maximum scalability and performance.

III. Case Study: Modeling SPECjAppServer2004

In this section, we present a detailed case study with SPECjAppServer2004 which demonstrates our approach for modeling J2EE applications and predicting their performance under load. Note that in [13] and [14], we modeled previous versions of the benchmark using Queueing Petri Net (QPN) models and Queueing Network (QN) models, respectively. In both cases, we ran against some serious problems stemming from the size and complexity of the system modeled. These problems were addressed in [15] by means of SimQPN - our tool for analyzing QPN models using simulation. In this paper, we present another application of SimQPN, this time analyzing QPN models of SPECjAppServer2004. However, the models considered here span the whole benchmark application and are much more complex and sophisticated. Thanks to the increased modeling accuracy and representativeness, the models demonstrate much better predictive power and scalability than what was achieved in our previous work. The case study presented here is the first comprehensive validation of our modeling approach on a *complete end-to-end* DCS, representative of today's real-life systems.

In our validation, we consider multiple scenarios varying the transaction mix and workload intensity. To achieve this, we had to modify the SPECjAppServer2004 driver to make it possible to precisely configure the transaction mix injected. Therefore, since we use a modified version of the driver, we would like to note that the SPECjAppServer2004 tests conducted as part of this study are **not** comparable to standard SPECjAppServer2004 results. Moreover, since our results have not been reviewed by SPEC, any comparison or performance inference against published benchmark results is strictly prohibited.

A. Motivation

Consider an automobile manufacturing company that wants to exploit e-business technology to support its order-inventory, supply-chain and manufacturing operations. The company has decided to use the J2EE platform and is in the process of developing a J2EE application. Lets assume that the first prototype of this application is SPECjAppServer2004 and that the company is testing the application in the deployment environment depicted in Figure 2. This environment uses a cluster of WebLogic servers (WLS) as a J2EE container and an Oracle database server (DBS) for persistence. A third-party HTTP load balancer is employed to distribute the load over the WebLogic servers. We assume that all servers in the WebLogic cluster are identical and that initially only two servers are available.

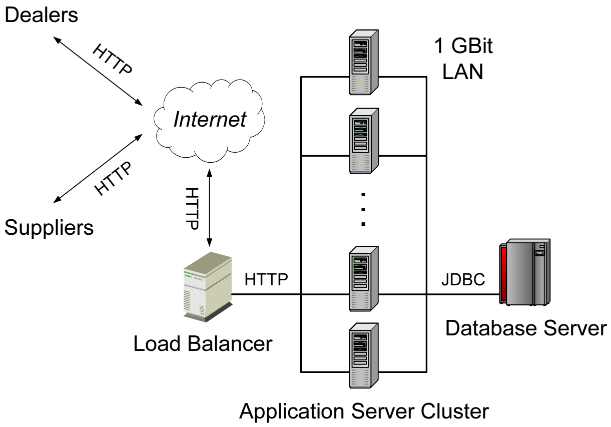


Fig. 2. SPECjAppServer2004 deployment environment.

The company is now about to conduct a performance evaluation of their system in order to find answers to the following questions:

- For a given number of WebLogic servers, what level of performance would the system provide?
- How many WebLogic servers would be needed to guarantee adequate performance under the expected workload?
- Will the capacity of the single load balancer and single database server suffice to handle the incoming load?
- Does the system scale or are there any other potential system bottlenecks?

The following sections show how these questions can be addressed by means of our performance modeling and prediction approach.

B. Performance Modeling and Prediction Approach

Our approach builds on the methodologies proposed by Menascé, Almeida and Dowdy in [16]–[20], however, a major difference is that our approach is based on Queueing Petri Net (QPN) models as opposed to conventional Queueing Network (QN) models and it is specialized for distributed component-based systems. QPN models are more sophisticated than QN models and enjoy greater modeling power and expressiveness. As shown in [13], QPN models lend themselves very well to modeling J2EE applications and provide a number of advantages over conventional QN models.

The modeling process includes the following steps:

1) Establish performance modeling objectives.
2) Characterize the system in its current state.
3) Characterize the workload.
4) Develop a performance model.
5) Validate, refine and/or calibrate the model.
6) Use model to predict system performance.
7) Analyze results and address modeling objectives.

It is important to note that the modeling process is iterative in nature and the above steps might have to be repeated multiple times as the system and workload evolve. In the following sections, we go through each of the seven steps and show how they are applied to our scenario. For a more general discussion of the methodology refer to [21].

C. Step 1: Establish performance modeling objectives.

Some general goals of the modeling study were listed above. At the beginning of the modeling process, these goals need to be made more specific and precise. Lets assume that under normal operating conditions the company expects to have 72 concurrent dealer clients (40 Browse, 16 Purchase and 16 Manage) and 50 planned production lines. During peak conditions, 152 concurrent dealer clients (100 Browse, 26 Purchase and 26 Manage) are expected and the number of planned production lines could increase up to 100. Moreover, the workload is forecast to grow by 300% over the next 5 years. The average *dealer think time* is 5 seconds, i.e. the time a dealer ”thinks” after receiving a response from the system before sending a new request. On average 10% of all orders placed are assumed to be large orders. The average delay after completing a work order at a planned production line before starting a new one is 10 seconds. Note that all of these numbers were chosen arbitrarily in order to make our motivating scenario more specific. Based on these assumptions the following concrete goals are established:

- Predict the performance of the system under normal operating conditions with 4 and 6 WebLogic servers, respectively. What would be the average transaction throughput, transaction response time and server CPU utilization?
- Determine if 6 WebLogic servers would be enough to ensure that the average response times of business transactions do not exceed half a second during peak conditions.
- Predict how much system performance would improve if the load balancer is upgraded with a slightly faster CPU.
- Study the scalability of the system as the workload increases and additional WebLogic servers are added.
- Determine which servers would be most utilized under heavy load and investigate if they are potential bottlenecks.

D. Step 2: Characterize the system in its current state.

In this step, the system is described in detail, in terms of its hardware and software architecture. The goal here is to obtain an in-depth understanding of the system architecture and its components. The latter is essential for building representative models.

As shown in Figure 2, the system we are considering has a two-tier hardware architecture consisting of an application server tier and a database server tier. Incoming requests are evenly distributed across the nodes in the application server cluster. For HTTP requests (dealer application) this is achieved using a software load balancer running on a dedicated machine. For RMI requests (manufacturing application) this is done transparently by the EJB client stubs. The application logic is partitioned into three layers: presentation layer (Servlets/JSPs), business logic layer (EJBs) and data layer (DBMS). Table I describes the system components in terms of the hardware and software platforms used. This information is enough for the purposes of our study.

TABLE I
SYSTEM COMPONENT DETAILS

Component	Description
Load Balancer	HTTP load balancer 1 x AMD Athlon XP2000+ CPU 1 GB RAM, SuSE Linux 8
App. Server Cluster Nodes	WebLogic 8.1 Server 1 x AMD Athlon XP2000+ CPU 1 GB RAM, SuSE Linux 8
Database Server	Oracle 9i Server 2 x AMD Athlon MP2000+ CPU 2 GB RAM, SuSE Linux 8
Local Area Network	1 GBit Switched Ethernet

E. Step 3: Characterize the workload.

In this step, the workload of the system under study is described in a qualitative and quantitative manner. This is called *workload characterization* and includes five major steps [16], [22]:

1) Identify the basic components of the workload.
2) Partition basic components into workload classes.
3) Identify the system components and resources (hardware and software) used by each workload class.
4) Describe the inter-component interactions and processing steps for each workload class.
5) Characterize workload classes in terms of their service demands and workload intensity.

We now go through each of these steps and apply it to our scenario.

1) Identify the basic components of the workload: As discussed in Section II, the SPECjAppServer2004 benchmark application is made up of three major subapplications - the dealer application in the dealer domain, the order entry application in the customer domain and the manufacturing application in the manufacturing domain. The dealer and order entry applications process business transactions of three types - Browse, Purchase and Manage. Hereafter, the latter are referred to as *dealer transactions* [3]. The manufacturing application, on the other hand, is running production lines which process work orders. Thus, the SPECjAppServer2004 workload is composed of two basic components: dealer transactions and work orders.

Note that each dealer transaction emulates a client session comprising multiple round-trips to the server. For each round-trip there is a separate HTTP request which can be seen as a subtransaction. A more fine-grained approach to model the workload would be to define the individual HTTP requests as basic components. However, this would unnecessarily complicate the workload model since we are interested in the performance of dealer transactions as a whole and not in the performance of their individual subtransactions. The same reasoning applies to work orders because each work order comprises multiple JTA transactions initiated with separate RMI calls. This is a typical example how the level of detail

[3] The term transaction here is used loosely and should not be confused with a database transaction or a transaction in the sense of the Java Transaction API (JTA transaction).

in the modeling process is decided based on the modeling objectives.

2) Partition basic components into workload classes: The basic components of real workloads are typically heterogeneous in nature. In order to improve the representativeness of the workload model and increase its predictive power, the basic components must be partitioned into classes (called *workload classes*) that have similar characteristics. The partitioning can be done based on different criteria depending on the type of system modeled and the goals of the modeling effort [18], [23].

We now partition the basic components of the workload into classes according to the type of work being done. There are three types of dealer transactions - Browse, Purchase and Manage. Since we are interested in their individual behavior, we model them using separate workload classes. Work orders, on the other hand, can be divided into two types based on whether they are processed on a planned or large order line. While planned lines run on a predefined schedule, large order lines run only when a large order arrives in the customer domain. Each large order generates a separate work order processed *asynchronously* on a dedicated large order line. Thus, work orders originating from large orders are different from ordinary work orders in terms of the way their processing is initiated and in terms of their resource usage. To distinguish between the two types of work orders they are modeled using two separate workload classes: *WorkOrder* (for ordinary work orders) and *LargeOrder* (for work orders generated by large orders). The latter will hereafter be referred to as WorkOrder and LargeOrder transactions, respectively. Altogether, we end up with five workload classes: Browse, Purchase, Manage, WorkOrder and LargeOrder.

3) Identify the system components and resources used by each workload class: The next step is to identify the system components (hardware and software resources) used by each workload class. The following hardware resources are used by dealer transactions:

- The CPU of the load balancer machine (LB-C)
- The CPU of an application server in the cluster (AS-C)
- The CPUs of the database server (DB-C)
- The disk drive of the database server (DB-D)
- The Local Area Network (LAN)

WorkOrders and LargeOrders use the same resources with exception of the first one (LB-C), since their processing is driven through direct RMI calls to the EJBs in the WebLogic cluster bypassing the HTTP load balancer. As far as software resources are concerned, all workload classes use the WebLogic servers and the Oracle DBMS. Dealer transactions additionally use the software load balancer which is running on a dedicated server. For a transaction to be processed by a WebLogic server, a thread must be allocated from the server's thread pool.

4) Describe the inter-component interactions and processing steps for each workload class: The aim of this step is to describe the flow of control for each workload class. Also for each processing step, the hardware and software resources used must be specified. Different notations may be exploited for this purpose, for example Client/Server Interaction Diagrams (CSID) [19], Communication-Processing Delay Diagrams [18], Execution Graphs [24], as well as conventional UML Sequence and Activity Diagrams [25].

All of the five workload classes identified comprise multiple processing tasks and therefore we will refer to them as composite transactions. Figures 3 and 4 use execution graphs to illustrate the processing steps (subtransactions) of transactions from the different workload classes. For every subtransaction, multiple system components are involved and they interact to perform the respective operation. The inter-component interactions and flow of control for subtransactions are depicted in Figure 5 by means of Client/Server Interaction Diagrams (CSID) [19]. Directed arcs show the flow of control from one node to the next during execution. Depending on the path followed, different execution scenarios are possible. For example, for dealer subtransactions two scenarios are possible depending on whether the database needs to be accessed or not. Dealer subtransactions that do not access the database (e.g. goToHomePage) follow the path $1 \to 2 \to 3 \to 4$, whereas dealer subtransactions that access the database (e.g. showInventory or checkOut) follow the path $1 \to 2 \to 3 \to 5 \to 6 \to 7$. Since most dealer subtransactions do access the database, for simplicity, it is assumed that all of them follow the second path.

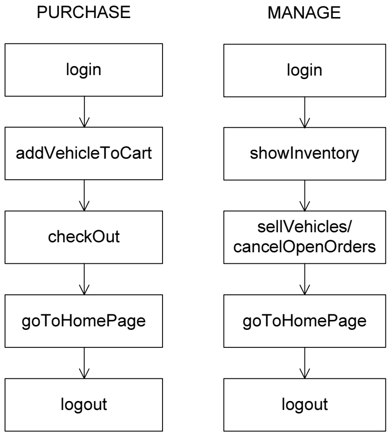


Fig. 3. Execution graphs showing the subtransactions of Purchase and Manage transactions.

5) Characterize workload classes in terms of their service demands and workload intensity: In this step, the load placed by the workload classes on the system is quantified. Two sets of parameters must be specified for each workload class: service demand parameters and workload intensity parameters. Service demand parameters specify the total amount of service time required by each workload class at each resource. Workload intensity parameters provide for each workload class a measure of the number of units of work, i.e. requests or transactions, that contend for system resources.

Since the system is available for testing, the service demands can be determined by injecting load into the system and taking measurements. Note that it is enough to have a single WebLogic server available in order to do this. The *Service Demand Law* can be used to estimate the service demands based

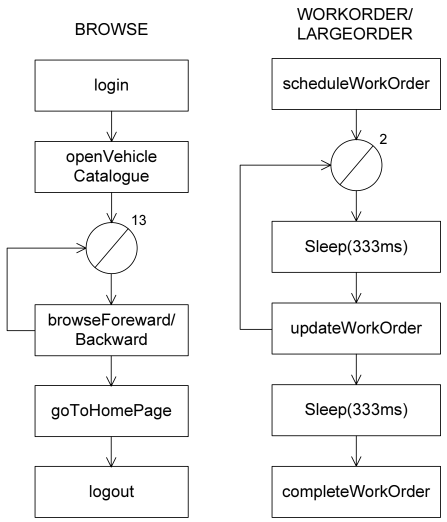
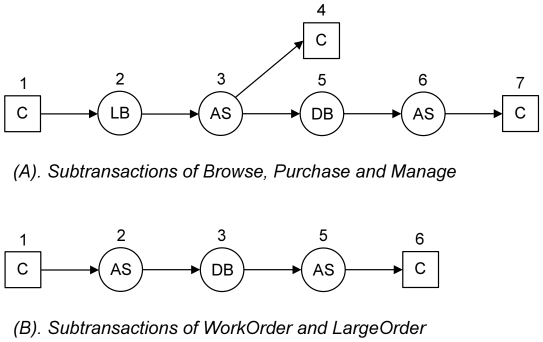


Fig. 4. Execution graphs showing the subtransactions of Browse, WorkOrder and LargeOrder transactions.



(A). Subtransactions of Browse, Purchase and Manage

(B). Subtransactions of WorkOrder and LargeOrder

Fig. 5. Client/Server interaction diagrams showing the flow of control during processing of subtransactions.

on measured resource utilization and transaction throughput data [26]. The latter states that the service demand $D_{i,r}$ of class r transactions at resource i is equal to the average utilization $U_{i,r}$ of resource i by class r transactions, divided by the average throughput $X_{0,r}$ of class r transactions during the measurement interval, i.e.

$$D_{i,r} = \frac{U_{i,r}}{X_{0,r}} \tag{1}$$

For each of the five workload classes a separate experiment was conducted measuring the utilization of the various system resources. CPU utilization was measured using the `vmstat` utility on Linux. The disk utilization of the database server was measured with the help of the Oracle 9i Intelligent Agent which proved to have negligible overhead. Table II reports the estimated service demand parameters for the five request classes in our workload model. It was decided to ignore the network, since all communications were taking place over 1 GBit LAN and communication times were negligible.

Note that, in order to keep the workload model simple, it is assumed that the total service demand of a transaction at a given system resource is spread evenly over its subtransactions. Thus, the service demand of a subtransaction can be estimated by dividing the measured total service demand of the transaction by the number of subtransactions it has. Whether this simplification is acceptable will become clear later when the model is validated. In case the estimation proves to be too inaccurate, we would have to come back and refine the workload model by measuring the service demands of subtransactions individually.

TABLE II

WORKLOAD SERVICE DEMAND PARAMETERS

Workload Class	LB-C	AS-C	DB-C	DB-D
Browse	42.72ms	130ms	14ms	5ms
Purchase	9.98ms	55ms	16ms	8ms
Manage	9.93ms	59ms	19ms	7ms
WorkOrder	-	34ms	24ms	2ms
LargeOrder	-	92ms	34ms	2ms

TABLE III

WORKLOAD INTENSITY PARAMETERS

Parameter	Normal Conditions	Peak Conditions
Browse Clients	40	100
Purchase Clients	16	26
Manage Clients	16	26
Planned Lines	50	100
Dealer Think Time	5 sec	5 sec
Mfg Think Time	10 sec	10 sec

Now that the service demands of workload classes have been quantified, the workload intensity must be specified. For each workload class, the number of units of work (transactions) that contend for system resources must be indicated. The way workload intensity is specified is dictated by the modeling objectives. In our case, workload intensity was defined in terms of the following parameters (see Section III-C):

- Number of concurrent dealer clients of each type and the average dealer think time.
- Number of planned production lines and the average time they wait after processing a WorkOrder before starting a new one (*manufacturing think time* or *mfg think time*).

With workload intensity specified in this way, all workload classes are automatically modeled as closed. Two scenarios of interest were indicated when discussing the modeling objectives in Section III-C, operation under normal conditions and operation under peak conditions. The values of the workload intensity parameters for these two scenarios are shown in Table III. However, the workload had been forecast to grow by 300% and another goal of the study was to investigate the scalability of the system as the load increases. Therefore, scenarios with up to 300% higher workload intensity need to be considered as well.

F. Step 4: Develop a performance model.

In this step, a performance model is developed that represents the different components of the system and its workload, and captures the main factors affecting its performance. The performance model can be used to understand the behavior of the system and predict its performance under load. The approach presented here exploits QPN models to take advantage of the modeling power and expressiveness of QPNs. For an introduction to the QPN formalism refer to [13], [21], [27].

We start by discussing the way basic components of the workload are modeled. During workload characterization, five workload classes were identified. All of them represent composite transactions and are modeled using the following token types (colors): 'B' for Browse, 'P' for Purchase, 'M' for Manage, 'W' for WorkOrder and 'L' for LargeOrder.

The subtransactions of transactions from the different classes were shown in Figures 3 and 4. The inter-component interactions and flow of control for each subtransaction were illustrated in Figure 5. In order to make the performance model more compact, it is assumed that each server used during processing of a subtransaction is visited only once and that the subtransaction receives all of its service demands at the server's resources during that single visit. This simplification is typical for queueing models and has been widely employed. Similarly, during the service of a subtransaction at a server, for each server resource used (e.g. CPUs, disk drives), it is assumed that the latter is visited only one time receiving the whole service demand of the subtransaction at once. These simplifications make it easier to model the flow of control during processing of subtransactions. While characterizing the workload service demands in Section III-E.5, we additionally assumed that the total service demand of a transaction at a given system resource is spread evenly over its subtransactions. This allows us to consider the subtransactions of a given workload class as equivalent in terms of processing behavior and resource consumption. Thus, we can model subtransactions using a single token type (color) per workload class as follows: 'b' for Browse, 'p' for Purchase, 'm' for Manage, 'w' for WorkOrder and 'l' for LargeOrder. For the sake of compactness, the following additional notation will be used:

Symbol 'D' will be used to denote a 'B', 'P' or 'M' token, i.e. token representing a dealer transaction.

Symbol 'd' will be used to denote a 'b', 'p' or 'm' token, i.e. token representing a dealer subtransaction.

Symbol 'o' will be used to denote a 'b', 'p', 'm', 'w' or 'l' token, i.e. token representing a subtransaction of arbitrary type, hereafter called *subtransaction token*.

To further simplify the model, we assume that LargeOrder transactions are executed with a single subtransaction, i.e. their four subtransactions are bundled into a single subtransaction. The effect of this simplification on the overall system behavior is negligible, because large orders constitute only 10% of all orders placed, i.e. relatively small portion of the system workload. Following these lines of thought one could consider LargeOrder transactions as non-composite and drop the small 'l' tokens. However, in order to keep token definitions uniform across transaction classes, we will keep the small 'l' tokens and look at LargeOrder transactions as being composed of a single subtransaction represented by an 'l' token.

Following the guidelines for modeling the system components, resources and inter-component interactions presented in [21], we arrive at the model depicted in Figure 6. We use the notation "$A\{x\} \rightarrow B\{y\}$" to denote a firing mode in which an 'x' token is removed from place A and a 'y' token is deposited in place B. Similarly, "$A\{x\} \rightarrow \{\}$" means that an 'x' token is removed from place A and destroyed without depositing tokens anywhere. Table IV provides some details on the places used in the model.

All token service times at the queues of the model are assumed to be exponentially distributed. We now examine in detail the life-cycle of tokens in the QPN model. As already discussed, upper-case tokens represent transactions, whereas lower-case tokens represent subtransactions. In the initial marking, tokens exist only in the depositories of places C_1 and C_2. The initial number of 'D' tokens ('B', 'P' or 'M') in the depository of the former determines the number of concurrent dealer clients, whereas the initial number of 'W' tokens in the depository of the latter determines the number of planned production lines running in the manufacturing domain. When a dealer client starts a dealer transaction, transition t_1 is fired destroying a 'D' token from the depository of place C_1 and creating a 'd' token in place G, which corresponds to starting the first subtransaction. The flow of control during processing of subtransactions in the system is modeled by moving their respective subtransaction tokens across the different places of the QPN. Starting at place G, a dealer subtransaction token ('d') is first sent to place L where it receives service at the CPU of the load balancer. After that it is moved to place E and from there it is routed to one of the N application server CPUs represented by places A_1 to A_N. Transitions $t_{11}, \ t_{13}, \ldots, t_{10+N}$ have equal firing probabilities (weights), so that subtransactions are probabilistically load-balanced across the N application servers. This approximates the round-robin mechanism used by the load-balancer to distribute incoming requests among the servers. Having completed its service at the application server CPU, the dealer subtransaction token is moved to place F from where it is sent to one of the two database server CPUs with equal probability (transitions t_4 and t_5 have equal firing weights). After completing its service at the CPU, the dealer subtransaction token is moved to place H where it receives service from the database disk subsystem. Once this is completed, the dealer subtransaction token is destroyed by transition t_8 and there are two possible scenarios:

1) A new 'd' token is created in place G, which starts the next dealer subtransaction.
2) If there are no more subtransactions to be executed, the 'D' token removed from place C_1 in the beginning of the transaction is returned. If the completed transaction is of type Purchase and it has generated a large order, additionally a token 'l' is created in place E.

After a 'D' token of a completed transaction returns back to place C_1, it spends some time at the IS queue of the latter. This corresponds to the time the dealer client "thinks" before starting the next transaction. Once the dealer think time has elapsed, the 'D' token is moved to the depository and the next transaction is started.

When a WorkOrder transaction is started on a planned line in the manufacturing domain, transition t_0 is fired destroying a 'W' token from the depository of place C_2 and creating

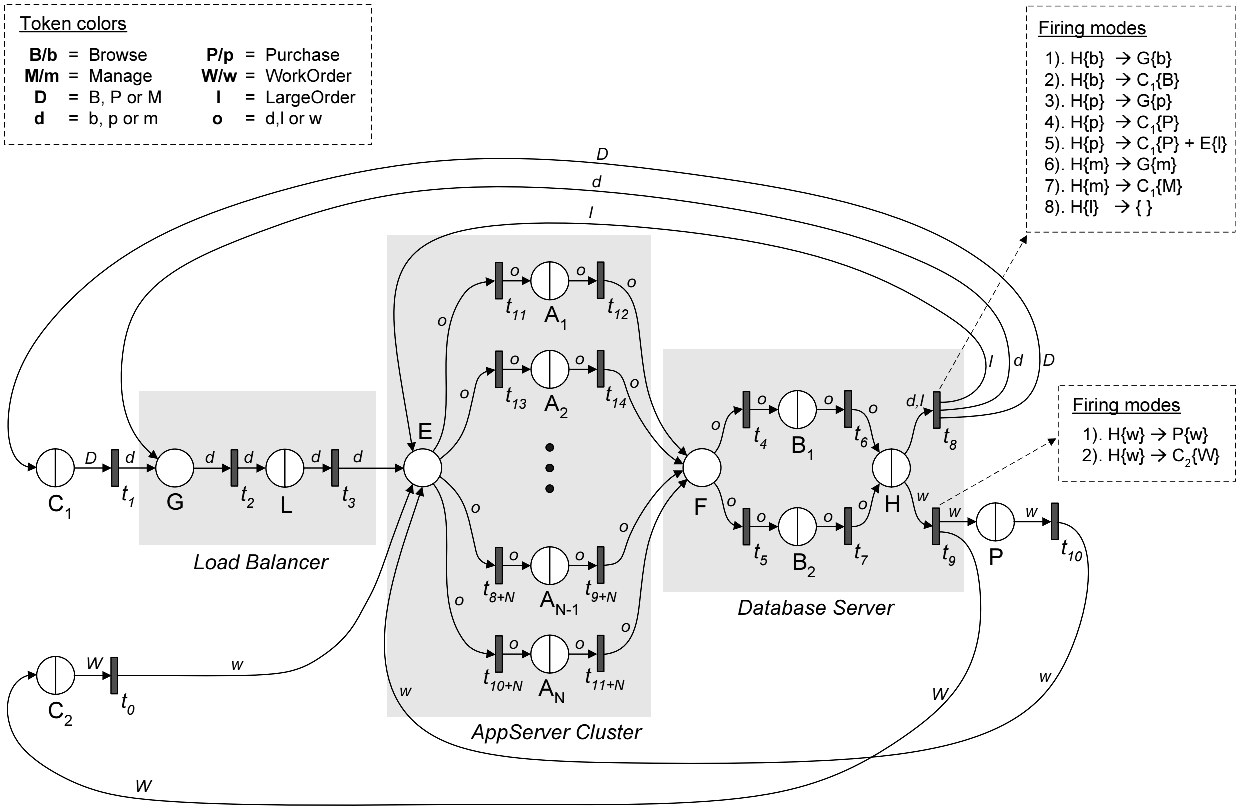


Fig. 6. Queueing Petri Net model of the system.

TABLE IV
PLACES USED IN THE QUEUEING PETRI NET MODEL.

Place	Tokens	Queue Type	Description
C_1	{B,P,M}	$G/M/\infty/IS$	Queueing place used to model concurrent dealer clients conducting dealer transactions. The time tokens spend here corresponds to the dealer think time.
C_2	{W}	$G/M/\infty/IS$	Queueing place used to model planned production lines driving work order processing. The time tokens spend here corresponds to the mfg think time.
G	{b,p,m}	*na*	Ordinary place where dealer subtransaction tokens are created when new subtransactions are started.
L	{b,p,m}	$G/M/1/PS$	Queueing place used to model the CPU of the load balancer machine.
E	{b,p,m,l,w}	*na*	Ordinary place where subtransaction tokens arrive before they are distributed over the application server nodes.
A_i	{b,p,m,l,w}	$G/M/1/PS$	Queueing places used to model the CPUs of the N application server nodes.
F	{b,p,m,l,w}	*na*	Ordinary place where subtransaction tokens arrive when visiting the database server. From here tokens are evenly distributed over the two database server CPUs.
B_j	{b,p,m,l,w}	$G/M/1/PS$	Queueing places used to model the two CPUs of the database server.
H	{b,p,m,l,w}	$G/M/1/FCFS$	Queueing place used to model the disk subsystem (made up of a single 100 GB disk drive) of the database server.
P	{w}	$G/M/\infty/IS$	Queueing place used to model the virtual production line stations that work orders move along during their processing. The time tokens spend here corresponds to the average delay at a production line station (i.e. 333 ms) emulated by the manufacturing application.

a ’w’ token in place E, which corresponds to starting the first subtransaction. Since WorkOrder subtransaction requests are load-balanced transparently (by the EJB client stubs) without using a load balancer, the WorkOrder subtransaction token (’w’) is routed directly to the application server CPUs - places A_1 to A_N. It then moves along the places representing the application server and database server resources exactly in the same way as dealer subtransaction tokens. After it completes its service at place H, the following two scenarios are possible:

1) The ’w’ token is sent to place P whose IS queue delays it for 333 ms, corresponding to the delay at a virtual production line station. After that the token is destroyed by transition t_{10} and a new ’w’ token is created in place E, representing the next WorkOrder subtransaction.

TABLE V
FIRING MODES OF TRANSITION t_8

Mode	Action	Case Modeled
1	$H\{b\} \rightarrow G\{b\}$	Browse subtransaction has been completed. Parent transaction is not finished yet.
2	$H\{b\} \rightarrow C_1\{B\}$	Browse subtransaction has been completed. Parent transaction is finished.
3	$H\{p\} \rightarrow G\{p\}$	Purchase subtransaction has been completed. Parent transaction is not finished yet.
4	$H\{p\} \rightarrow C_1\{P\}$	Purchase subtransaction has been completed. Parent transaction is finished.
5	$H\{p\} \rightarrow C_1\{P\} + E\{l\}$	Same as (4), but assuming that completed transaction has generated a large order.
6	$H\{m\} \rightarrow G\{m\}$	Manage subtransaction has been completed. Parent transaction is not finished yet.
7	$H\{m\} \rightarrow C_1\{M\}$	Manage subtransaction has been completed. Parent transaction is finished.
8	$H\{l\} \rightarrow \{\}$	LargeOrder transaction has been completed. Its token is simply destroyed.

2) If there are no more subtransactions to be executed, the 'w' token is destroyed by transition t_9 and the 'W' token removed from place C_2 in the beginning of the transaction is returned.

After a 'W' token of a completed transaction returns back to place C_2, it spends some time at the IS queue of the latter. This corresponds to the time waited after completing a work order at a production line before starting the next one. Once this time has elapsed, the 'W' token is moved to the depository and the next transaction is started.

All transitions of the model are immediate and, with exception of t_8 and t_9, they all have equal weights for all of their firing modes. The assignment of firing weights to transitions t_8 and t_9 is critical to achieving the desired behavior of transactions in the model. Weights must be assigned in such a way that transactions are terminated only after all of their subtransactions have been completed. We will now explain how this is done starting with transition t_9 which has the following two firing modes:

1) $H\{w\} \rightarrow P\{w\}$: Corresponds to the case where a WorkOrder subtransaction has been completed, but its parent transaction is not finished yet. The parent transaction is delayed for 333 ms at the production line station (place P) and then its next subtransaction is started by depositing a new 'w' token in place E.
2) $H\{w\} \rightarrow C_2\{W\}$: Corresponds to the case where a WorkOrder subtransaction has been completed leading to completion of its parent transaction. The 'W' token removed from place C_2 in the beginning of the parent WorkOrder transaction is now returned back.

According to Section III-E.4 (Figure 4), WorkOrder transactions are comprised of four subtransactions. This means that, for every WorkOrder transaction, four subtransactions have to be executed before the transaction is completed. To model this behavior, the firing weights (probabilities) of modes 1 and 2 are set to $3/4$ and $1/4$, respectively. Thus, out of every four times a 'w' token arrives in place H and enables transition t_9, on average the latter will be fired three times in mode 1 and one time in mode 2, completing a WorkOrder transaction.

Transition t_8, on the other hand, has eight firing modes as shown in Table V. According to Section III-E.4 (Figure 3), Browse transactions have 17 subtransactions, whereas Purchase and Manage have only 5. This means that, for every Browse transaction, 17 subtransactions have to be executed before the transaction is completed, i.e. out of every 17 times a 'b' token arrives in place H and enables transition t_8, the latter has to be fired 16 times in mode 1 and one time in mode 2 completing a Browse transaction. Similarly, out of every 5 times an 'm' token arrives in place H and enables transition t_8, the latter has to be fired 4 times in mode 6 and one time in mode 7 completing a Manage transaction. Out of every 5 times a 'p' token arrives in place H and enables transition t_8, the latter has to be fired 4 times in mode 3 and one time in mode 4 or mode 5, depending on whether a large order has been generated. On average 10% of all completed Purchase transactions generate large orders. Modeling these conditions probabilistically leads to a system of simultaneous equations that the firing weights (probabilities) of transition t_8 need to fulfil. One possible solution is the following: $w(1) = 16$, $w(2) = 1$, $w(3) = 13.6$, $w(4) = 3.06$, $w(5) = 0.34$, $w(6) = 13.6$, $w(7) = 3.4$, $w(8) = 17$.

The workload intensity and service demand parameters from Section III-E.5 (Tables II and III) are used to provide values for the service times of tokens at the various queues of the model. A separate set of parameter values is specified for each workload scenario considered. The service times of subtransactions at the queues of the model are estimated by dividing the total service demands of the respective transactions by the number of subtransactions they have.

G. *Step 5: Validate, refine and/or calibrate the model.*

TABLE VI
INPUT PARAMETERS FOR VALIDATION SCENARIOS

Parameter	Scenario 1	Scenario 2
Browse Clients	20	40
Purchase Clients	10	20
Manage Clients	10	30
Planned Lines	30	50
Dealer Think Time	5 sec	5 sec
Mfg Think Time	10 sec	10 sec

The model developed in the previous sections is now validated by comparing its predictions against measurements on the real system. Two application server nodes are available for the validation experiments. The model predictions are verified for a number of different scenarios under different transaction mixes and workload intensities. The model input parameters for two specific scenarios considered here are

TABLE VII
VALIDATION RESULTS

	Validation Scenario 1			Validation Scenario 2		
METRIC	Model	Measured	Error	Model	Measured	Error
Browse Throughput	3.784	3.718	+1.8%	6.988	6.913	+1.1%
Purchase Throughput	1.948	1.963	-0.7%	3.781	3.808	-0.7%
Manage Throughput	1.940	1.988	-2.4%	5.634	5.530	+1.9%
WorkOrder Throughput	2.713	2.680	+1.2%	4.469	4.510	-0.9%
LargeOrder Throughput	0.197	0.214	-8.1%	0.377	0.383	-1.56%
Browse Response Time	289ms	256ms	+12.9%	704ms	660ms	+6.7%
Purchase Response Time	131ms	120ms	+9.2%	309ms	305ms	+1.3%
Manage Response Time	139ms	130ms	+6.9%	329ms	312ms	+5.4%
WorkOrder Response Time	1087ms	1108ms	-1.9%	1199ms	1209ms	-0.8%
Load Balancer CPU Utilization	20.1%	19.5%	+3.1%	39.2%	40.3%	-2.7%
WebLogic Server CPU Utilization	41.3%	38.5%	+7.3%	81.8%	83.0%	-1.4%
Database Server CPU Utilization	9.7%	8.8%	+10.2%	19.3%	19.3%	0.0%

shown in Table VI. Table VII compares the model predictions against measurements on the system. The maximum modeling error for throughput is 8.1%, for utilization 10.2% and for response time 12.9%. Varying the transaction mix and workload intensity led to predictions of similar accuracy. Since these results are reasonable, the model is considered valid. Note that the model validation process is iterative in nature and is usually repeated multiple times as the model evolves. Even though the model is deemed valid at this point of the study, the model might lose its validity when it is modified in order to reflect changes in the system. Generally, it is required that the validation is repeated after every modification of the model.

H. Step 6: Use model to predict system performance.

In Section III-C some concrete goals were set for the performance study. The system model is now used to predict the performance of the system for the deployment configurations and workload scenarios of interest. In order to validate our approach, for each scenario considered we compare the model predictions against measurements on the real system. Note that this validation is not part of the methodology itself and normally it does not have to be done. Indeed, if we would have to validate the model results for every scenario considered, there would be no point in using the model in the first place. The reason we validate the model results here is to demonstrate the effectiveness of our methodology and showcase the predictive power of the QPN models it is based on.

Table IX reports the analysis results for the scenarios under normal operating conditions with 4 and 6 application server nodes. In both cases, the model predictions are very close to the measurements on the real system. Even for response times, the modeling error does not exceed 10.1%. Table X shows the model predictions for two scenarios under peak conditions with 6 application server nodes. The first one uses the original load balancer, while the second one uses an upgraded load balancer with a faster CPU. The faster CPU results in lower service demands as shown in Table VIII. With the original load balancer six application server nodes turned out to be insufficient to guarantee average response times of business transactions below half a second. However, with the upgraded load balancer this was achieved. In the rest of the scenarios considered, the upgraded load balancer will be used.

TABLE VIII
LOAD BALANCER SERVICE DEMANDS

Load Balancer	Browse	Purchase	Manage
Original	42.72ms	9.98ms	9.93ms
Upgraded	32.25ms	8.87ms	8.56ms

We now investigate the behavior of the system as the workload intensity increases beyond peak conditions and further application server nodes are added. Table XI shows the model predictions for two scenarios with an increased number of concurrent Browse clients, i.e. 150 in the first one and 200 in the second one. In both scenarios the number of application server nodes is 8. As evident from the results, the load balancer is completely saturated when increasing the workload intensity and it becomes a bottleneck limiting the overall system performance. Therefore, adding further application server nodes would not bring any benefit, unless the load balancer is replaced with a faster one.

I. Step 7: Analyze results and address modeling objectives.

We can now use the results from the performance analysis to address the goals established in Section III-C. By means of the developed QPN model, we were able to predict the performance of the system under normal operating conditions with 4 and 6 WebLogic servers. It turned out that using the original load balancer six application server nodes were insufficient to guarantee average response times of business transactions below half a second. Upgrading the load balancer with a slightly faster CPU led to the CPU utilization of the load balancer dropping by a good 20 percent. As a result, the response times of dealer transactions improved by 15 to 27 percent meeting the ”half a second” requirement. However, increasing the workload intensity beyond peak conditions revealed that the load balancer was a bottleneck resource, preventing us to scale the system by adding additional WebLogic

TABLE IX

ANALYSIS RESULTS FOR SCENARIOS UNDER NORMAL CONDITIONS WITH 4 AND 6 APP. SERVER NODES

	4 App. Server Nodes			6 App. Server Nodes		
METRIC	Model	Measured	Error	Model	Measured	Error
Browse Throughput	7.549	7.438	+1.5%	7.589	7.415	+2.3%
Purchase Throughput	3.119	3.105	+0.5%	3.141	3.038	+3.4%
Manage Throughput	3.111	3.068	+1.4%	3.117	2.993	+4.1%
WorkOrder Throughput	4.517	4.550	-0.7%	4.517	4.320	+4.6%
LargeOrder Throughput	0.313	0.318	-1.6%	0.311	0.307	+1.3%
Browse Response Time	299ms	282ms	+6.0%	266ms	267ms	-0.4%
Purchase Response Time	131ms	119ms	+10.1%	116ms	110ms	+5.5%
Manage Response Time	140ms	131ms	+6.9%	125ms	127ms	-1.6%
WorkOrder Response Time	1086ms	1109ms	-2.1%	1077ms	1100ms	-2.1%
Load Balancer CPU Utilization	38.5%	38.0%	+1.3%	38.7%	38.5%	+0.1%
WebLogic Server CPU Utilization	38.0%	35.8%	+6.1%	25.4%	23.7%	+0.7%
Database Server CPU Utilization	16.7%	18.5%	-9.7%	16.7%	15.5%	+0.8%

TABLE X

ANALYSIS RESULTS FOR SCENARIOS UNDER PEAK CONDITIONS WITH 6 APP. SERVER NODES

	Original Load Balancer			Upgraded Load Balancer		
METRIC	Model	Measured	Error	Model	Measured	Error
Browse Throughput	17.960	17.742	+1.2%	18.471	18.347	+0.7%
Purchase Throughput	4.981	4.913	+1.4%	5.027	5.072	-0.8%
Manage Throughput	4.981	4.995	-0.3%	5.013	5.032	-0.4%
WorkOrder Throughput	8.984	8.880	+1.2%	9.014	8.850	+1.8%
LargeOrder Throughput	0.497	0.490	+1.4%	0.501	0.515	-2.7%
Browse Response Time	567ms	534ms	+6.2%	413ms	440ms	-6.5%
Purchase Response Time	214ms	198ms	+8.1%	182ms	165ms	+10.3%
Manage Response Time	224ms	214ms	+4.7%	193ms	187ms	+3.2%
WorkOrder Response Time	1113ms	1135ms	-1.9%	1115ms	1123ms	-0.7%
Load Balancer CPU Utilization	86.6%	88.0%	-1.6%	68.2%	70.0%	-2.6%
WebLogic Server CPU Utilization	54.3%	53.8%	+0.9%	55.4%	55.3%	+0.2%
Database Server CPU Utilization	32.9%	34.5%	-4.6%	33.3%	35.0%	-4.9%

TABLE XI

ANALYSIS RESULTS FOR SCENARIOS UNDER HEAVY LOAD WITH 8 APP. SERVER NODES

	Heavy Load Scenario 1			Heavy Load Scenario 2		
METRIC	Model	Measured	Error	Model	Measured	Error
Browse Throughput	26.505	25.905	+2.3%	28.537	26.987	+5.7%
Purchase Throughput	4.948	4.817	+2.7%	4.619	4.333	+6.6%
Manage Throughput	4.944	4.825	+2.5%	4.604	4.528	+1.6%
WorkOrder Throughput	8.984	8.820	+1.8%	9.003	8.970	+0.4%
LargeOrder Throughput	0.497	0.488	+1.8%	0.460	0.417	+10.4%
Browse Response Time	664ms	714ms	-7.0%	2012ms	2288ms	-12.1%
Purchase Response Time	253ms	257ms	-1.6%	632ms	802ms	-21.2%
Manage Response Time	263ms	276ms	-4.7%	630ms	745ms	-15.4%
WorkOrder Response Time	1116ms	1128ms	-1.1%	1123ms	1132ms	-0.8%
Load Balancer CPU Utilization	94.1%	95.0%	-0.9%	99.9%	100.0%	-0.1%
WebLogic Server CPU Utilization	54.5%	54.1%	+0.7%	57.3%	55.7%	+2.9%
Database Server CPU Utilization	38.8%	42.0%	-7.6%	39.6%	42.0%	-5.7%

servers (see Figure 7). Therefore, in light of the expected workload growth, the company should either replace the load balancer machine with a faster one or consider using a more efficient load balancing method. After this is done, the performance analysis should be repeated to make sure that there are no other system bottlenecks.

IV. CONCLUSION

In this paper, we presented a case study with the industry-standard SPECjAppServer2004 benchmark in which the latter was used as a representative J2EE application in order to introduce a systematic approach for predicting the performance and scalability of J2EE applications. The approach is based on performance modeling and simulation. It takes

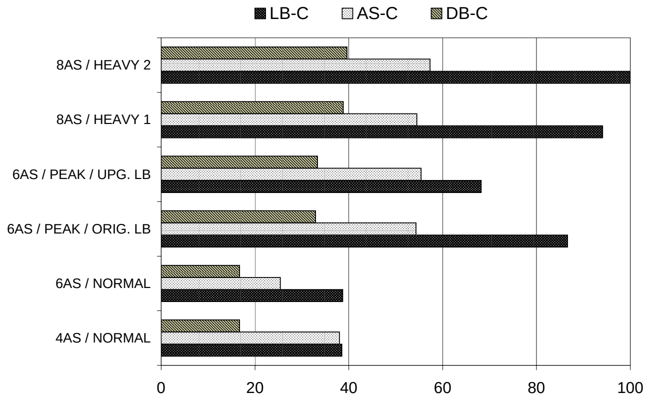


Fig. 7. Predicted server CPU utilization in considered scenarios.

advantage of the modeling power and expressiveness of the QPN formalism to improve model representativeness and enable accurate performance prediction. A detailed model of a SPECjAppServer2004 deployment was built in a step-by-step fashion. The model representativeness was validated by comparing its predictions against measurements on the real system. A number of different deployment configurations and workload scenarios were considered. In addition to CPU and I/O contention, it was shown how some more complex aspects of system behavior such as composite transactions and asynchronous processing can be modeled. The model proved to accurately reflect the performance and scalability characteristics of the system under study. The modeling error for transaction response time did not exceed 21.2% and was much lower for transaction throughput and resource utilization. The proposed performance modeling methodology provides a powerful tool for performance prediction that can be used in the software engineering process of J2EE applications.

Acknowledgments

The author would like to acknowledge the support of SPEC's co-founder and former President Kaivalya Dixit (in memoriam) during the development and specification of the SPECjAppServer2004 benchmark. SPECial thanks also to my friends and colleagues from SPEC's OSG Java Subcommittee, especially to John Stecher from IBM, Tom Daly and Akara Sucharitakul from Sun Microsystems and Ricardo Morin from Intel.

References

[1] S. Kounev, "SPECjAppServer2004 - The New Way to Evaluate J2EE Performance," DEV2DEV Article, O'Reilly Publishing Group, 2005, http://www.dev2dev.com.

[2] Standard Performance Evaluation Corporation (SPEC), "SPECjAppServer2004 Documentation, Specifications," Apr. 2004, http://www.spec.org/jAppServer2004/.

[3] IBM Software, "Trade3 Web Application Server Benchmark," 2003, http://www-306.ibm.com/software/webservers/appserv/benchmark3.html.

[4] Sun Microsystems, Inc., "Java^{TM}Pet Store Demo," 2003, http://java.sun.com/developer/releases/petstore/.

[5] .NET Framework Community (GotDotNet), "Microsoft .NET vs. Sun Microsystem's J2EE: The Nile E-Commerce Application Server Benchmark," Oct. 2001, http://www.gotdotnet.com/team/compare/nileperf.aspx, http://www.gotdotnet.com/team/compare/Nile

[6] Urbancode, Inc., "Urbancode EJB Benchmark," 2002, http://www.urbancode.com/projects/ejbbenchmark/default.jsp.

[7] ObjectWeb Consortium, "RUBiS: Rice University Bidding System," 2003, http://rubis.objectweb.org/.

[8] ——, "Stock-Online Project," 2003, http://forge.objectweb.org/projects/stock-online/.

[9] S. Kounev, B. Weis, and A. Buchmann, "Performance Tuning and Optimization of J2EE Applications on the JBoss Platform," *Journal of Computer Resource Management*, vol. 113, 2004.

[10] N. Chalainanont, E. Nurvitadhi, K. Chow, and S. L. Lu, "Characterization of L3 Cache Bahavior of Java Application Server," in *Proceedings of the 7th Workshop on Computer Architecture Evaluation using Commercial Workloads*, Feb. 2004.

[11] I. Gorton and A. Liu, "Performance Evaluation of EJB-Based Component Architectures," *IEEE Internet Computing*, vol. 7, no. 3, pp. 18–23, May 2003.

[12] S. Kounev and A. Buchmann, "Improving Data Access of J2EE Applications by Exploiting Asynchronous Processing and Caching Services," in *Proceedings of the 28th International Conference on Very Large Data Bases - VLDB2002, Hong Kong, China, August 20-23*, 2002.

[13] ——, "Performance Modelling of Distributed E-Business Applications using Queuing Petri Nets," in *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software - ISPASS2003, Austin, Texas, USA, March 20-22*, 2003.

[14] ——, "Performance Modeling and Evaluation of Large-Scale J2EE Applications," in *Proceedings of the 29th International Conference of the Computer Measurement Group (CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems - CMG2003, Dallas, TX, USA, December 7-12*, 2003.

[15] ——, "SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation," *Performance Evaluation*, 2005, to appear.

[16] D. A. Menascé, V. A. Almeida, and L. W. Dowdy, *Capacity Planning and Performance Modeling - From Mainframes to Client-Server Systems*. Prentice Hall, Englewood Cliffs, NG, 1994.

[17] D. Menascé, V. Almeida, R. Fonseca, and M. Mendes, "A Methodology for Workload Characterization of E-commerce Sites," in *Proceedings of the 1st ACM conference on Electronic commerce, Denver, Colorado, United States*, Nov. 1999, pp. 119–128.

[18] D. Menascé and V. Almeida, *Capacity Planning for Web Performance: Metrics, Models and Methods*. Prentice Hall, Upper Saddle River, NJ, 1998.

[19] ——, *Scaling for E-Business - Technologies, Models, Performance and Capacity Planning*. Prentice Hall, Upper Saddle River, NJ, 2000.

[20] D. A. Menascé, V. A. Almeida, and L. W. Dowdy, *Performance by Design*. Prentice Hall, 2004.

[21] S. Kounev, "Performance Engineering of Distributed Component-Based Systems - Benchmarking, Modeling and Performance Prediction," Ph.D. Thesis, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany, Aug. 2005.

[22] M. Calzarossa and G. Serazzi, "Workload Characterization: a Survey," *Proceedings of the IEEE*, vol. 8, no. 81, pp. 1136–1150, 1993.

[23] J. Mohr and S. Penansky, "A forecasting oriented workload characterization methodology," *CMG Transactions*, vol. 36, June 1982.

[24] C. U. Smith and L. G. Williams, *Performance Solutions - A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[25] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[26] P. Denning and J. Buzen, "The Operational Analysis of Queueing Network Models," *ACM Computing Surveys*, vol. 10, no. 3, pp. 225–261, Sept. 1978.

[27] F. Bause and F. Kritzinger, *Stochastic Petri Nets - An Introduction to the Theory*, 2nd ed. Vieweg Verlag, 2002.