# S

## SOFTWARE PERFORMANCE EVALUATION

### INTRODUCTION

Performance and quality of service (QoS) aspects of modern software systems are crucially important for their successful adoption in the industry. Most generally, the *performance* of a software system indicates the degree to which the system meets its objectives for timeliness and the efficiency with which it achieves this. Timeliness is normally measured in terms of meeting certain response time or throughput requirements and scalability goals. Response time refers to the time required to respond to a user request, for example a Web service call or a database transaction, and throughput refers to the number of requests or jobs processed per unit of time. Scalability, on the other hand, is understood as the ability of the system to continue to meet its objectives for response time and throughput as the demand for the services it provides increases and resources (typically hardware) are added.

Numerous studies, for example, exist in the areas of e-business, manufacturing, telecommunications, military, health care, and transportation that have shown that a failure to meet the performance requirements can lead to serious financial losses, loss of customers and reputation, and in some cases even to loss of human lives. To avoid the pitfalls of inadequate QoS, it is important to evaluate the expected performance characteristics of systems during all phases of their lifecycle. The methods used to do this are part of the discipline called *software performance engineering* (SPE) (1,2). Software performance engineering helps to estimate the level of performance a system can achieve and provides recommendations to realize the optimal performance level (3).

However, as systems grow in size and complexity, estimating their performance becomes a more and more challenging task. Modern software systems are often composed of multiple components deployed in highly distributed and heterogeneous environments. Figure 1 shows a typical architecture of a multitiered distributed component-based system (4). The application logic is partitioned into components distributed over physical tiers. Three tiers exist: presentation tier, business logic tier, and data tier. The presentation tier includes Web servers hosting Web components that implement the presentation logic of the application. The business logic tier includes a cluster of application servers hosting business logic components that implement the business logic of the application. Middleware platforms such as Java EE (5), Microsoft .NET (6), or CORBA (7) are often used in this tier to simplify application development by leveraging some common services typically used in enterprise applications. The data tier includes database servers and legacy systems that provide data management services. The inherent complexity of such architectures makes it difficult to manage their end-to-end performance and scalability. To avoid performance problems, it is essential that systems are subjected to rigorous performance evaluation during the various stages of their lifecycle. At every stage, performance evaluation is conducted with a specific set of goals and constraints. The goals can be classified in the following categories, some of which partially overlap:

**Platform selection:** Determine which hardware and software platforms would provide the best scalability and cost/performance ratio. Software platforms include operating systems, middleware, database management systems, and so on. Hardware platforms include the type of servers, disk subsystems, load balancers, communication networks, and so on.

**Platform validation:** Validate a selected combination of platforms to ensure that taken together they provide adequate performance and scalability.

**Evaluation of design alternatives:** Evaluate the relative performance and scalability of alternative system designs and architectures.

**Performance prediction:** Predict the performance of the system for a given workload and configuration scenario.

**Performance tuning:** Analyze the effect of various deployment settings and tuning parameters on the system performance and find their optimal values.

**Performance optimization:** Find the components with the largest effect on performance and study the performance gains from optimizing them.

**Scalability and bottleneck analysis:** Study the performance of the system as the load increases and more hardware is added. Find which system components are most utilized and investigate whether they are potential bottlenecks.

**Sizing and capacity planning:** Determine the amount of hardware that would be needed to guarantee certain performance levels.

Two broad approaches help conduct performance evaluation of software systems: *performance measurement* and *performance modeling*. In the first approach, load testing tools and benchmarks are used to generate artificial workloads on the system and to measure its performance. In the second approach, performance models are built and then used to analyze the performance and scalability characteristics of the system. In both cases, it is necessary to characterize the workload of the system under study before performance evaluation can be conducted. The *workload* can be defined as the set of all inputs that the system receives from its environment during a period of time (3).



*Wiley Encyclopedia of Computer Science and Engineering*, edited by Benjamin Wah.
Copyright © 2008 John Wiley & Sons, Inc.

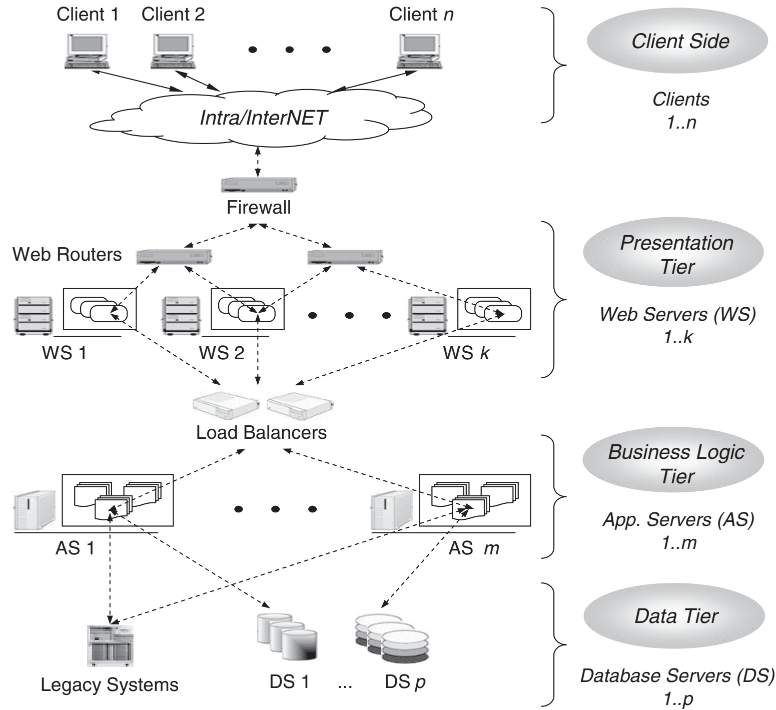


**Figure 1.** A multitiered distributed component-based system.

In performance evaluation studies normally workload models are used that are representations of the real system workloads.

## WORKLOAD MODELS

A *workload model* is a representation that captures the main aspects of the real workload that have effect on the performance measures of interest. We distinguish between *executable* and *nonexecutable* models. Executable models are programs that mimic real workloads and can be used to evaluate the system performance in a controlled environment. For example, an executable model could be a set of benchmark programs that emulate real users sending requests to the system. Nonexecutable models, on the other hand, are abstract workload descriptions normally used as input to analytical or simulation models of the system. For example, a nonexecutable model could be a set of parameter values that describe the types of requests processed by the system and their load intensities. Workload models are aimed to be as compact as possible and at the same time representative of the real workloads under study.

As shown in Fig. 2, workload models can be classified into two major categories: *natural models* and *artificial models* (3). Natural models are constructed from real workloads of the system under study or from execution traces of real workloads. In the former case, they are called *natural benchmarks,* and in the latter case they are called *workload traces*. A natural benchmark is a set of programs extracted from the real workload such that they represent the major characteristics of the latter. A workload trace is a chronological sequence of records describing specific events that were observed during execution of the real workload. For example, in the three-tier environment described earlier, the logs collected by the servers at each tier (Web servers, application servers and database servers) can be used as workload traces. Although traces usually exhibit good representativeness, they have the drawback that they normally consist of huge amounts of data and do not provide a compact representation of the workload.

Unlike natural models, artificial workload models are not constructed using basic components of real workloads as building blocks, however, they try to mimic the real workloads. Artificial models can be classified into *synthetic benchmarks, application benchmarks*, and *abstract*

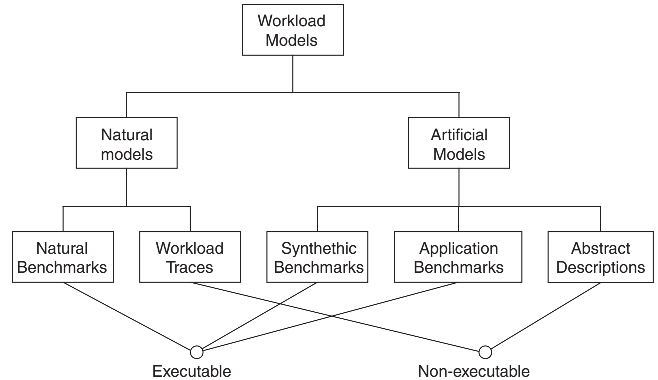


**Figure 2.** Taxonomy of workload models.

*workload descriptions*. Synthetic benchmarks are artificial programs carefully chosen to match the relative mix of operations observed in some class of applications. They usually do no real, useful work. In contrast, application benchmarks are complete real-life applications. They are normally designed specifically to be representative of a given class of applications. Finally, abstract workload descriptions are nonexecutable models composed of a set of parameter values that characterize the workload in terms of the load it places on the system components. Such models are typically used in conjunction with analytical or simulation models of the system. Depending on the type of workload, different parameters may be used, such as transaction/request types, times between successive request arrivals (interarrival times), transaction execution rates, transaction service times at system resources, and so on. As an example, an e-commerce workload can be described by specifying the types of requests processed by the system (e.g., place order, change order, cancel order), the rates at which requests arrive, and the amount of resources used when processing requests, that is, the time spent receiving service at the various system resources such as central processing units (CPUs), input–output (I/O) devices, and networks. For additional examples and details on executable and nonexecutable workload models, the reader is referred to Refs. 8 and 9, as well as 3 and 10, respectively.

## PERFORMANCE MEASUREMENT

The measurement approach to software performance evaluation is typically applied in three contexts:

– *Platform benchmarking:* Measure the performance and scalability of alternative platforms on which a system can be built and/or deployed.
– *Application profiling:* Measure and profile the performance of application components during the various stages of the development cycle.
– *System load testing:* Measure the end-to-end system performance under load in the later stages of development when a running implementation or a prototype is available for testing.

In all three cases, executable workload models are used. In this section, we briefly discuss the above three contexts in which performance measurements are done. A more detailed introduction to performance measurement techniques can be found in Refs. 1, 8, 9, 11 and 12. The *Proceedings of the Annual Conference of the Computer Measurement Group* (CMG) are an excellent source of recent publications on performance measurement tools, methodologies, and concepts.

### Platform Benchmarking

While benchmarking efforts have traditionally been focused on hardware performance, over the past 15 years, benchmarks have increasingly been used to evaluate the performance and scalability of end-to-end systems including both the hardware and *software* platforms used to build them (9). Thus, the scope of benchmarking efforts has expanded to include software products like Web servers, application servers, database management systems, message-oriented middleware, and virtual machine monitors. Building on scalable and efficient platforms is crucial to achieving good performance and scalability. Therefore, it is essential that platforms are validated to ensure that they provide adequate level of performance and scalability before they are used to build real applications. Where alternative platforms are available, benchmark results can be used for performance comparisons to help select the platform that provides the best cost/performance ratio. Two major benchmark standardization bodies exist, the Standard Performance Evaluation Corporation (SPEC) (13) and the Transaction Processing Performance Council (TPC) (14). Many standard benchmarks have appeared in the last decade that provide means to measure the performance and scalability of software platforms. For example, SPECjAppServer2004 and TPC-App for application servers, SPECjbb2005 for server-side Java, TPC-W and SPECweb2005 for Web servers, TPC-C, TPC-E and TPC-H for database management systems, and SPECjms2007 for message-oriented middleware. Benchmarks such as these are called *application benchmarks* because they are designed to be representative of a given class of real-world applications.

Although the main purpose of application benchmarks is to measure the performance and scalability of alternative platforms on which a system can be built, they can also be used to study the effect of platform configuration settings and tuning parameters on the overall system performance (9,15,16). Thus, benchmarking not only helps to select platforms and validate their performance and scalability, but also helps to tune and optimize the selected platforms for optimal performance. The *Proceedings of the Annual SPEC Benchmark Workshops* are an excellent source on the latest developments in benchmarking methodologies and tools (17).

### Application Profiling

Application profiling is conducted iteratively during the system development cycle to evaluate the performance of components as they are designed and implemented. Design and implementation decisions taken at the early stages of system development are likely to have a strong impact on the overall system performance (1). Moreover, problems caused by poor decisions taken early in the development cycle are usually most expensive and time-consuming to correct. Therefore, it is important that, as components are designed and implemented, their performance is measured and profiled to ensure that they do not have any internal bottlenecks or processing inefficiencies. Software profilers are normally used for this purpose.

*Software profilers* are performance measurement tools that help to gain a comprehensive understanding of the execution-time behavior of software components. They typically provide information such as the fraction of time spent in specific states (e.g., executing different subroutines, blocking on I/O, running operating system kernel

code) and the flow of control during program execution. Two general techniques are normally used to obtain such information, *statistical sampling* and *code instrumentation* (11,12). The statistical sampling approach is based on interrupting the program execution periodically and recording the execution state. The code instrumentation approach, on the other hand, is based on modifying the program code to record state information whenever a specified set of events of interest occur. Statistical sampling is usually much less intrusive; however, it only provides statistical summary of the times spent in different states and cannot provide any information on how the various states were reached (e.g., call graphs). Code instrumentation, on the other hand, is normally more intrusive; however, it allows the profiler to precisely record all the events of interest as well as the call sequences that show the flow of control during program execution. For example, in CPU time profiling, statistical sampling may reveal the relative percentage of time spent in frequently-called methods, whereas code instrumentation can report the exact number of times each method is invoked and the calling sequence that led to the method invocation.

### System Load Testing

Load testing is typically done in the later stages of system development when a running implementation or a prototype of the system is available for testing. Load-testing tools are used to generate synthetic workloads and measure the system performance under load. Sophisticated load-testing tools can emulate hundreds of thousands of "virtual users" that mimic real users interacting with the system. While tests are run, system components are monitored and performance metrics (e.g., response time, throughput and utilization) are measured. Results obtained in this way can be used to identify and isolate system bottlenecks, fine-tune system components, and measure the end-to-end system scalability (18). Unfortunately, this approach has several drawbacks. First of all, it is not applicable in the early stages of system development when the system is not available for testing. Second, it is extremely expensive and time-consuming because it requires setting up a production-like testing environment, configuring load testing tools, and conducting the tests. Finally, testing results normally cannot be reused for other applications.

## PERFORMANCE MODELING

The performance modeling approach to software performance evaluation is based on using mathematical or simulation models to predict the system performance under load. Models represent the way system resources are used by the workload and capture the main factors that determine the system behavior under load (10). This approach is normally much cheaper than load testing and has the advantage that it can be applied in the early stages of system development before the system is available for testing. A number of different methods and techniques have been proposed in the literature for modeling software systems and predicting their performance under load. Most of them, however, are based on the same general methodology that proceeds through the steps depicted in Fig. 3 (1, 3, 19–21).

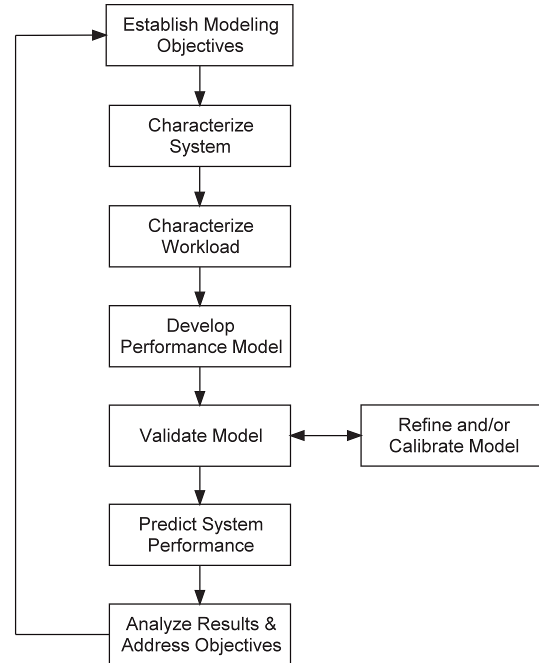


**Figure 3.** Performance modeling process.

First, the goals and objectives of the modeling study are specified. After this, the system is described in detail in terms of its hardware and software architecture. The aim is to obtain an in-depth understanding of the system architecture and its components. Next, the workload of the system is characterized and a workload model is built. The workload model is used as a basis to develop a performance model. Before the model can be used for performance prediction, it has to be validated. This is done by comparing performance metrics predicted by the model with measurements on the real system. If the predicted values do not match the measured values within an acceptable level of accuracy, then the model must be refined and/or calibrated. Finally, the validated performance model is used to predict the system performance for the deployment configurations and workload scenarios of interest. The model predictions are analyzed and used to address the goals set in the beginning of the modeling study. We now take a closer look at the major steps of the modeling process.

### Workload Characterization

Workload characterization is the process of describing the workload of the system in a qualitative and quantitative manner (20). The result of workload characterization is a nonexecutable workload model that can be used as input to performance models. Workload characterization usually involves the following activities (1, 22):

- The basic components of the workload are identified.
- Basic components are partitioned into workload classes.
- The system components/resources used by each workload class are identified.

– The inter-component interactions and processing steps are described.
– Service demands and workload intensities are quantified.

In the following, we discuss each of these activities in turn.

**The Basic Components of the Workload are Identified.** *Basic component* refers to a generic unit of work that arrives at the system from an external source (19). Some examples include HTTP requests, remote procedure calls, Web service invocations, database transactions, interactive commands, and batch jobs. Basic components could be composed of multiple processing tasks, for example client sessions that comprise multiple requests to the system or nested transactions (open or closed). The choice of basic components and the decision how granular they are defined depend on the nature of the services provided by the system and on the modeling objectives. Because, in almost all cases, basic components can be considered as some kind of requests or transactions processed by the system, they are often referred to as *requests* or *transactions*[1].

**Basic Components are Partitioned into Workload Classes.** To improve the representativeness of the workload model, the basic components are 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 (19, 23). The basic components should be partitioned in such a way that each workload class is as homogeneous as possible in terms of the load it places on the system and its resources.

**The System Components and Resources Used by Each Workload Class are Identified.** For example, an online request to place an order might require using a Web server, application server, and backend database server. For each server, the concrete hardware and software resources used must be identified. It is distinguished between active and passive resources (10). An *active resource* is a resource that delivers a certain service to transactions at a finite speed (e.g., CPU or disk drive). In contrast, a *passive resource* is needed for the execution of a transaction, but it is not characterized by a speed of service delivery (e.g., thread, database connection or main memory).

**The Intercomponent Interactions and Processing Steps are Described.** The aim of this step is to describe the processing steps, the inter-component interactions, and the flow of control for each workload class. Also for each processing step, the hardware and software resources used are specified. Different notations may be exploited for this purpose, for example client/server interaction diagrams (20), execution graphs (1), communication-processing delay diagrams (19), as well as conventional UML sequence and activity diagrams (24).

**Service Demands and Workload Intensities are Quantified.** The goal is to quantify the load placed by the workload components on the system. *Service-demand parameters* specify the average total amount of service time required by each workload class at each resource. Most techniques for obtaining service-demand parameters involve running the system or components thereof and taking measurements. Some techniques are also available that can be used to estimate service-demand parameters in the early stages of system development before the system is available for testing (25). *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. Depending on the way workload intensity is specified, it is distinguished between open and closed classes. For open classes, workload intensity is specified as an arrival rate, whereas for closed classes it is specified as average number of requests served concurrently in the system.

The product of the workload characterization steps described above (i.e., the workload model) is sometimes referred to as *software execution model* because it represents the key facets of software execution behavior (1).

### Performance Models

A performance model is an abstract representation of the system that relates the workload parameters with the system configuration and captures the main factors that determine the system performance. Performance models can be used to understand the behavior of the system and predict its performance under load. Figure 4 shows the major types of performance models that are available in the literature for modeling computer systems. Note that this model classification is not clear cut because some model types partially overlap. Performance models can be grouped into two main categories: *simulation models* and *analytical models*. One of the greatest challenges in building a good model is to find the right level of detail. A general rule of thumb is: “Make the model as simple as possible, but not simpler!” Including too much detail might render the model intractable, on the other hand, making it too simple might render it unrepresentative.

**Simulation Models.** Simulation models are software programs that mimic the behavior of a system as requests arrive and get processed at the various system resources. Such models are normally stochastic because they have one or more random variables as input (e.g., the request interarrival times). The structure of a simulation program is based on the states of the simulated system and events that cause the system state to change. When implemented, simulation programs count events and record the duration of time spent in different states. Based on these data, performance metrics of interest (e.g., the average time a request takes to complete or the average system throughput) can be estimated at the end of the simulation run. Estimates

[1]The term transaction here is used loosely to refer to any unit of work or processing task executed in the system.

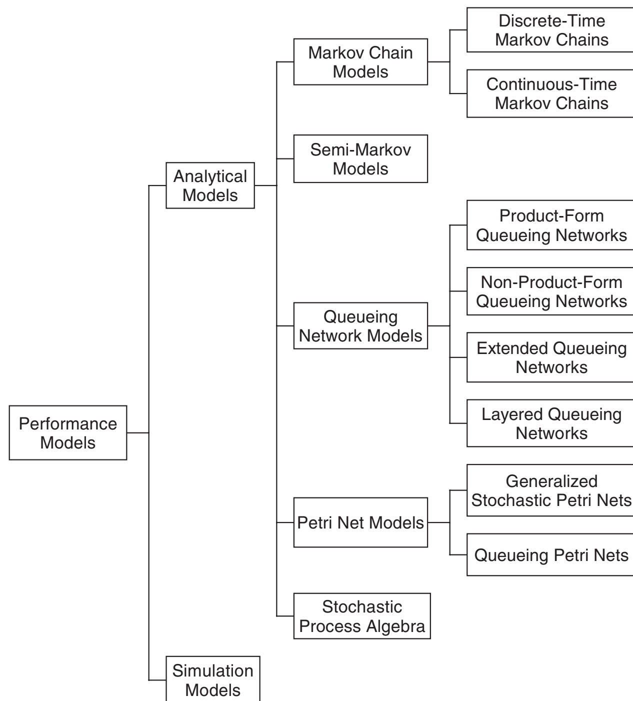


**Figure 4.** Major types of performance models.

are provided in the form of confidence intervals. A confidence interval is a range with a given probability that the estimated performance metric lies within this range. The main advantage of simulation models is that they are very general and can be made as accurate as desired. However, this accuracy comes at the cost of the time taken to develop and run the models. Usually, many long runs are required to obtain estimates of needed performance measures with reasonable confidence levels.

Several approaches to developing a simulation model (22) exist. The most time-consuming approach is to use a general purpose programming language such as C++ or Java, possibly augmented by simulation libraries (e.g., CSIM or SimPack). Another approach is to use a specialized simulation language such as GPSS/H, Simscript II.5, or MODSIM III. Finally, some simulation packages support graphical languages for defining simulation models (e.g., Arena, Extend, SES/workbench). A comprehensive treatment of simulation techniques can be found in Refs. 26 and 27.

**Analytical Models.** Analytical models are based on mathematical laws and computational algorithms used to derive performance metrics from model parameters. Analytical models are usually less expensive to build and more efficient to analyze compared with simulation models. However, because they are defined at a higher level of abstraction, they are normally less detailed and accurate. Moreover, for models to be mathematically tractable, usually many simplifying assumptions need to be made impairing the model representativeness.

Queueing networks and generalized stochastic Petri nets are perhaps the two most popular types of models used in practice. Queueing networks provide a very powerful mechanism for modeling hardware contention (contention for CPU time, disk access, and other hardware resources) and scheduling strategies. A number of efficient analysis methods have been developed for a class of queueing networks called *product-form queueing networks*, which enable models of realistic size and complexity to be analyzed (28). The downside of queueing networks is that they are not expressive enough to model software contention accurately (contention for processes, threads, database connections, and other software resources), as well as blocking, simultaneous resource possession, asynchronous processing, and synchronization aspects. Even though extensions of queueing networks, such as *extended queueing networks* (29) and *layered queueing networks* (also called stochastic rendezvous networks) (30–32), provide some support for modeling software contention and synchronization aspects, they are often restrictive and inaccurate.

In contrast to queueing networks, generalized stochastic Petri net models easily can express software contention, simultaneous resource possession, asynchronous processing, and synchronization aspects. Their major disadvantage, however, is that they do not provide any means for direct representation of scheduling strategies. The attempts to eliminate this disadvantage have led to the emergence of *queueing Petri nets* (33–35), which combine the modeling power and expressiveness of queueing networks and stochastic Petri nets. Queueing Petri nets enable the integration of hardware and software aspects of system behavior in the same model (36, 37). A major hurdle to the practical use of queueing Petri nets, however, is that their analysis suffers from the state space explosion problem limiting the size of the models that can be solved. Currently, the only way to circumvent this problem is by using simulation for model analysis (38).

Details of the various types of analytical models shown in Fig. 4 are beyond the scope of this article. The following books can be used as reference for additional information (3, 12, 28, 35, 39–42). The *Proceedings of the ACM SIGMETRICS Conferences* and the *Performance Evaluation Journal* report recent research results in performance modeling and evaluation. Further relevant information can be found in the *Proceedings of the International Conference on Quantitative Evaluation of SysTems (QEST)*, the *Proceedings of the Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (MASCOTS), the *Proceedings of the International Conference on Performance Evaluation Methodologies and Tools*, (VALUETOOLS) and the *Proceedings of the ACM International Workshop on Software and Performance* (WOSP).

## Model Validation and Calibration

Before a model can be used for performance prediction, it has to be validated. We assume that the system modeled or a prototype of it is available for testing. The model is said to be *valid* if the performance metrics (e.g., response time, throughput, and resource utilization) predicted by the model match the measurements on the real system within a certain acceptable margin of error (3). As a rule of thumb, errors within 10% for utilization and throughput, and

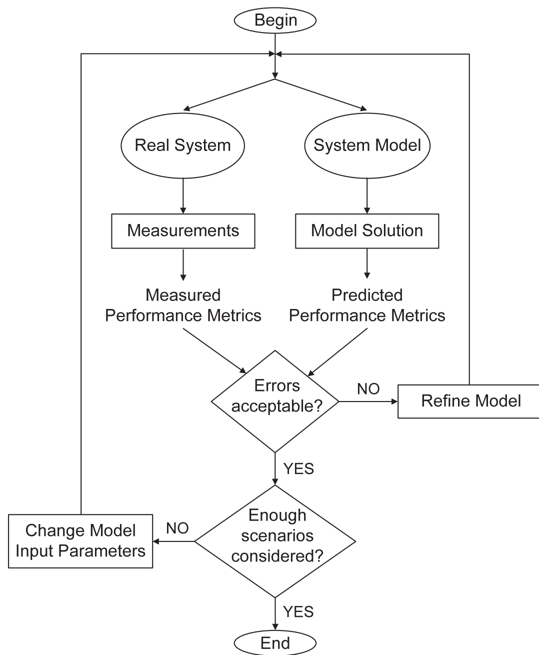


**Figure 5.** Model validation and refinement process.

within 20% for response time are considered acceptable (10). Model validation is normally conducted by comparing performance metrics predicted by the model with measurements on the real system. This testing is performed for several different scenarios varying the model input parameters. If the predicted values do not match the measured values within an acceptable level of accuracy, then the model must be refined. Otherwise, the model is deemed valid and can be used for performance prediction. The validation and refinement process is illustrated in Fig. 5. It is important that the model predictions are verified for several scenarios under different transaction mixes and workload intensities before the model is deemed valid. The model refinement process usually involves the following activities:

- The model input parameters are verified.
- Assumptions and simplifications made when building the model are revisited.
- The system is monitored under load to ensure that all critical aspects of its behavior are captured by the model.
- It is considered to increase the level of detail at which the system is modeled.

If after refining the model, predicted metrics still do not match the measurements on the real system within an acceptable level of accuracy, then the model has to be calibrated. *Model calibration* is the process of changing the model to force it to match the actual system (43). This is achieved by changing the values of some model input or output parameters. The parameters may be increased or decreased by an absolute or percentage amount. Normally, input parameters are changed (e.g., service demands); however, in certain cases, also output parameters might be changed. If an output parameter is altered when calibrating the baseline model, then it must be altered in the same manner whenever the model is used for performance prediction. After the model is calibrated, the validation procedure must be repeated to make sure that the calibrated model now accurately reflects the real system and workload. For a detailed discussion of model calibration techniques, the reader is referred to Refs. 10 and 44.

The extent to which a model can be validated quantitatively as described above depends on the availability of an implementation of the system components. In the initial phases of system development when no implementation is available, model validation would be limited to revisiting the assumptions made when building the model. If a system or a prototype with a similar architecture to the one modeled is available, then it could be used to provide some rough measurement data for quantitative validation.

### Software Performance Engineering

Over the last 15 years, a number of approaches have been proposed for integrating performance evaluation and prediction techniques into the software engineering process. Efforts were initiated with Smith's seminal work pioneered under SPE (45). Since then many meta-models for describing performance-related aspects (46) have been developed by the SPE community, the most prominent being the UML SPT profile and its successor the UML MARTE profile, both of which are extensions of UML as the de facto standard modeling language for software architectures. Other proposed meta-models include SPE-MM (47), CSM (48), and KLAPER (49). The common goal of these efforts is to enable the automated transformation of design-oriented software models into analysis-oriented performance models, which make it possible to predict the system performance. A recent servey of model-based performance prediction techniques was published in Ref. 50. Many techniques that use a range of different performance models have been proposed including standard queueing networks (3, 25, 47, 51), extended queueing networks (49, 52, 53), layered queueing networks (48), stochastic Petri nets (54, 55), and queueing Petri nets (4, 21). In recent years, with the increasing adoption of component-based software engineering, the performance evaluation community has focused on adapting and extending conventional SPE techniques to support component-based systems. For a recent survey of performance prediction methodologies and tools for component-based systems, refer to Ref. 56.

## OPERATIONAL ANALYSIS

An alternative approach to performance evaluation known as *operational analysis* is based on a set of basic invariant relationships between performance quantities (57). These relationships, which are commonly known as *operational laws*, can be considered as consistency requirements for the values of performance quantities measured in any particular experiment. We briefly present the most important operational laws. Consider a system made up of $K$ resources (e.g., servers, processors, disk drives, network links). The system processes transactions requested by clients. It is assumed that during the processing of a transaction,

multiple resources can be used and at each point in time the transaction is either being served at a resource or waiting for a resource to become available. A resource might be used multiple times during a transaction, and each time a request is sent to the resource, we will refer to this as the transaction *visiting* the resource. The following notation will be used:

$V_i$ the average number of times resource $i$ is visited during the processing of a transaction.
$S_i$ the average service time of a transaction at resource $i$ per visit to the resource.
$D_i$ the average total service time of a transaction at resource $i$.
$U_i$ the utilization of resource $i$ (i.e., the fraction of time the resource is busy serving requests).
$X_i$ the throughtput of resource $i$ (i.e., the number of service completions per unit time).
$X_0$ the system throughput (i.e., the number of transactions processed per unit time).
$R$ the average transaction response time (i.e., the average time it takes to process a transaction including both the waiting and service time in the system).
$N$ the average number of active transactions in the system, either waiting for service or being served.

If we observe the system for a finite amount of time $T$, assuming that the system is in steady state, then the following relationships can be shown to hold:

**Utilization Law:**

$$U_i = X_i \times S_i$$

**Forced Flow Law:**

$$X_i = X_0 \times V_i$$

**Service Demand Law:**

$$D_i = U_i / X_0$$

**Little's Law:**

$$N = X_0 \times R$$

The last of the above relationships, Little's Law, is one of the most important and fundamental laws in queueing theory. It can also be extended to higher moments (58). If we assume that transactions are started by a fixed set of $M$ clients and that the average time a client waits after completing a transaction before starting the next transaction (the client think time) is $Z$, then using Little's Law, the following relationship can be easily shown to hold:

**Interactive Response Time Law:**

$$R = \frac{M}{X_0} - Z$$

Although operational analysis is not as powerful as queueing theoretic methods for performance analysis, it has the advantage that it can be applied under much more general conditions because it does not require the strong assumptions typically made in stochastic modeling. For a more detailed introduction to operational analysis, the reader is referred to Refs. 3, 10, and 22.

## SUMMARY

In this article, an overview of the major methods and techniques for software performance evaluation was presented. First, the different types of workload models that are typically used in performance evaluation studies were considered. Next, an overview of common tools and techniques for performance measurement, including platform benchmarking, application profiling, and system load testing, was given. Then, the most common methods for workload characterization and performance modeling of software systems were surveyed. The major types of performance models used in practice were considered and their advantages and disadvantages were discussed. An outline of the approaches to integrating model-based performance analysis into the software engineering process was presented. Finally, operational analysis was introduced briefly as an alternative to queueing theoretic methods.

## BIBLIOGRAPHY

1. C. U. Smith and L. G. Williams, *Performance Solutions - A Practical Guide to Creating Responsive, Scalable Software*, Reading, MA: Addison-Wesley, 2002.
2. R. R. Dumke, C. Rautenstrauch, A. Schmietendorf, and A. Scholz, eds. *Performance Engineering, State of the Art and Current Trends, Vol. 2047 of Lecture Notes in Computer Science*, New York: Springer, 2001.
3. D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy, *Performance by Design*, Englewood Cliffs, NJ: Prentice Hall, 2004.
4. S. Kounev, *Performance Engineering of Distributed Component-Based Systems - Benchmarking, Modeling and Performance Prediction*, Herzogenrath, Germany: Shaker Verlag, 2005.
5. Sun Microsystems, Inc. Java Platform, Enterprise Edition (Java EE), 2007. http://java.sun.com/javaee/.
6. Microsoft Corp. Microsoft .NET Framework, 2007. http://msdn.microsoft.com/netframework/.
7. Object Management Group (OMG). Common Object Request Broker Architecture (CORBA), 2007. http://www.corba.org/.
8. L. K. John and L. Eeckhout, eds., *Performance Evaluation and Benchmarking*. Boca Raton, FL: CRC Press, 2006.
9. R. Eigenmann, ed., *Performance Evaluation and Benchmarking with Realistic Applications*. Cambridge, MA: The MIT Press, 2001.
10. D. A. Menascè, V. A. F. Almeida, and L. W. Dowdy, *Capacity Planning and Performance Modeling - From Mainframes to Client-Server Systems*, Englewood Cliffs, NJ: Prentice Hall, 1994.
11. D. Lilja, *Measuring Computer Performance: A Practitioner's Guide*, Cambridge, U.K., Cambridge University Press, 2000.

12. R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, New York: Wiley-Interscience, 1991.
13. Standard Performance Evaluation Corporation (SPEC). http://www.spec.org/.
14. Transaction Processing Performance Council (TPC). http://www.tpc.org/.
15. S. Kounev and A. Buchmann, Improving data access of J2EE applications by exploiting asynchronous processing and caching services, *Proc. of the 28th International Conference on Very Large Data Bases - VLDB2002*, Hong Kong, China, 2002.
16. S. Kounev, B. Weis, and A. Buchmann, Performance tuning and optimization of J2EE applications on the JBoss platform, *J. Comput. Res. Manage.*, **113**: 2004.
17. SPEC Benchmark Workshop Proceedings. http://www.spec.org/events/.
18. B. M. Subraya, *Integrated Approach to Web Performance Testing: A Practitioner's Guide*, Hershey, PA: IRM Press, 2006.
19. D. Menascé and V. Almeida, *Capacity Planning for Web Performance: Metrics, Models and Methods*, Upper Saddle River, NJ: Prentice Hall, 1998.
20. D. Menascé, V. Almeida, R. Fonseca, and M. Mendes, A methodology for workload characterization of e-commerce sites, *Proc. of the 1st ACM Conference on Electronic Commerce*, Denver, CO, 1999, pp. 119–128.
21. S. Kounev, Performance modeling and evaluation of distributed component-based systems using queueing Petri nets, *IEEE Trans. Soft. Engineer.*, **32**(7): 486–502, 2006.
22. D. Menascé and V. Almeida, *Scaling for E-Business - Technologies, Models, Performance and Capacity Planning*, Upper Saddle River, NJ: Prentice Hall, 2000.
23. J. Mohr and S. Penansky, A forecasting oriented workload characterization methodology, *CMG Trans.*, **36**: 1982.
24. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Reading, MA: Addison-Wesley, 1999.
25. D. A. Menascé and H. Gomaa, A method for desigh and performance modeling of client/server systems, *IEEE Trans. Soft. Engin.*, **26**(11): 2000.
26. A. Law and D. W. Kelton, *Simulation Modeling and Analysis*. 3rd ed. New York: Mc Graw Hill Companies, Inc., 2000.
27. J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol, *Discrete-Event System Simulation*, 3rd ed. Upper Saddle River, N.J: Prentice Hall, 2001.
28. G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*, 2nd ed. New York: John Wiley & Sons, Inc., 2006.
29. E. A. MacNair, An introduction to the research queueing package, *WSC '85: Proc. of the 17th Conference on Winter Simulation*, New York, NY, 1985, pp. 257–262.
30. M. Woodside, Tutorial Introduction to Layered Modeling of Software Performance, 3rd ed., 2000. Available: http://www.sce.carleton.ca/rads/lqn/lqn-documentation/tutorialg.pdf.
31. P. Maly and C. M. Woodside, Layered modeling of hardware and software, with application to a LAN extension router, *Proc. of the 11th International Conference on Computer Performance Evaluation Techniques and Tools - TOOLS 2000*, Motorola University, Schaumburg, *Ill*, 2000.
32. M. Woodside, J. Neilson, D. Petriu, and S. Majumdar, The stochastic rendezvous network model for performance of synchronous client-server-like distributed software, *IEEE Trans. Comput.*, **44**(1): 20–34, 1995.
33. F. Bause, Queueing Petri nets - A formalism for the combined qualitative and quantitative analysis of systems, *Proc. of the 5th International Workshop on Petri Nets and Performance Models*, Toulouse, France, 1993.
34. F. Bause and P. Buchholz, Queueing Petri nets with product form solution, *Perform. Eval.*, **32**(4): 265–299, 1998.
35. F. Bause and F. Kritzinger, *Stochastic Petri Nets - An Introduction to the Theory*, 2nd ed. New York: Vieweg Verlag, 2002.
36. F. Bause, P. Buchholz, and P. Kemper, Integrating software and hardware performance models using hierarchical queueing Petri nets, *Proc. of the 9. ITG / GI - Fachtagung Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, (MMB'97)*, Freiberg, Germany, 1997.
37. S. Kounev and A. Buchmann, Performance modelling of distributed e-business applications using queuing Petri nets, *Proc. of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software - ISPASS2003*, Austin, TX, 2003.
38. S. Kounev and A. Buchmann, SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation, *Perform. Eval.*, **63**(4-5): 364–394, 2006.
39. K. S. Trivedi, *Probability and Statistics with Reliability, Queueing and Computer Science Applications*, 2nd ed. New York: John Wiley & Sons, Inc., 2002.
40. R. Sahner, K. Trivedi, and A. Puliafito, *Performance and Reliability Analysis of Computer Systems - An Example-Based Approach Using the SHARPE Software Package*, Dordrecht, The Netherlands: Kluwer Academic Publishers, 1996.
41. K. Begain, G. Bolch, and H. Herold, *Practical Performance Modeling - Application of the MOSEL Language*, Dardrecht: The Netherlands, Kluwer Academic Publishers, 2001.
42. J. Hillston, *A Compositional Approach to Performance Modelling*, Cambridge, U.K., Cambridge University Press, 1996.
43. P. J. Buzen and A. W. Shum, Model calibration. in *Proc. of the 1989 International CMG Conference, Reno, Nevada*, 1989, pp. 808–811.
44. J. Flowers and L. W. Dowdy, A comparison of calibration techniques for queuing network models, *Proc. of the 1989 International CMG Conference*, Reno, Nevada, 1989 pp. 644–655.
45. C. U. Smith, *Performance Engineering of Software Systems*, Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1990.
46. V. Cortellessa, How far are we from the definition of a common software performance ontology? *WOSP'05: Proc. of the 5th international Workshop on Software and Performance*, 2005, pp. New York, NY195–204.
47. C. U. Smith, C. M. Llad, V. Cortellessa, A. Di Marco, and L. G. Williams, From UML models to software performance results: an SPE process based on XML interchange formats, *WOSP '05: Proc. of the 5th International Workshop on Software and Performance*, New York, NY, 2005, pp 87–95.
48. D. Petriu and M. Woodside, An intermediate metamodel with scenarios and resources for generating performance models from UML designs, *Soft. Sys. Mode.*, **6**(2): 163–184, 2007.
49. V. Grassi, R. Mirandola, and A. Sabetta, Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach, *J. Sys. Soft.*, **80**(4): 528–558, 2007.

50. S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, Model-based performance prediction in software development: A survey, *IEEE Trans. Soft. Engineer.*, **30**(5): 295–310, 2004.
51. V. S. Sharma, P. Jalote, and K. S. Trivedi, *A Performance Engineering Tool for Tiered Software Systems*. Los Alamitos, CA: IEEE Computer Society, 2006, pp. 63–70.
52. V. Cortellessa and R. Mirandola, Deriving a queueing network based performance model from UML diagrams, *WOSP '00: Proc. of the 2nd International Workshop on Software and Performance*, New York, NY, 2000, pp. 58–70.
53. A. D'Ambrogio and G. Iazeolla, Design of XMI-based tools for building EQN models of software systems, in P. Kokol (ed.), *IASTED International Conference on Software Engineering, part of the 23rd Multi-Conference on Applied Informatics*, Innsbruck, Austria, Calgary, Alberta, Canada: IASTED/ ACTA Press, 2005 pp. 366–371.
54. J. P. Lopez-Grao, J. Merseguer, and J. Campos, From UML activity diagrams to stochastic Petri nets: application to software performance engineering, *SIGSOFT Softw. Eng. Notes*, **29**(1): 25–36, 2004.
55. S. Bernardi and J. Merseguer, QoS assessment via stochastic analysis, *IEEE Inter. Comput.*, **10**(3): 32–42, 2006.
56. S. Becker, L. Grunske, R. Mirandola, and S. Overhage, Performance prediction of component-based systems: A survey from an engineering perspective, in R. H. Reussner, J. Stafford, and C. Szyperski, (eds.), *Architecting Systems with Trustworthy Components*, Vol. 3938 of *LNCS*, New York: Springer, 2006, pp. 169–192.
57. P. J. Denning and J. P. Buzen, The operational analysis of queueing network models, *ACM Comput. Surv.*, **10**(3): 225–261, 1978.
58. W. Witt, A review of L = lambda-W and extensions, *Queueing Sys.*, **9**(3): 235–268, 1991.

SAMUEL KOUNEV
University of Cambridge
Cambridge, United Kingdom