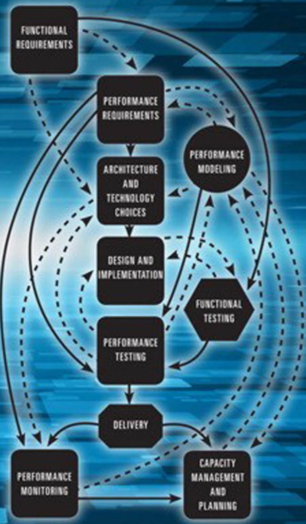
Foundations of Software and System Performance Engineering

Process, Performance Modeling, Requirements, Testing, Scalability, and Practice



André B. Bondi

Praise for *Foundations of Software and System Performance Engineering*

"If this book had only been available to the contractors building healthcare.gov, and they read and followed the lifecycle performance processes, there would not have been the enormous problems apparent in that health care application. In my 40-plus years of experience in building leading-edge software and hardware products, poor performance is the single most frequent cause of the failure or cancellation of a new, software-intensive project. Performance requirements are often neglected or poorly formulated during the planning and requirements phases of a project. Consequently, the software architecture and the resulting delivered system are unable to meet performance needs. This book provides the reader with the techniques and skills necessary to implement performance engineering at the beginning of a project and manage those requirements throughout the lifecycle of the product. I cannot recommend this book highly enough."

— *Don Shafer, CSDP, Technical Fellow, Athens Group, LLC*

"Well written and packed with useful examples, *Foundations of Software and System Performance Engineering* provides a thorough presentation of this crucial topic. Drawing upon decades of professional experience, Dr. Bondi shows how the principles of performance engineering can be applied in many different fields and disciplines."

— *Matthew Scarpino, author of* Programming the Cell Processor *and* OpenCL in Action

This page intentionally left blank

Foundations of Software and System Performance Engineering

This page intentionally left blank

Foundations of Software and System Performance Engineering

Process, Performance Modeling, Requirements, Testing, Scalability, and Practice

André B. Bondi

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Bondi, André B., author.
Foundations of software and system performance engineering : process, performance modeling, requirements, testing, scalability, and practice / André B. Bondi.
pages cm
Includes bibliographical references and index.
ISBN 978-0-321-83382-2 (pbk. : alk. paper)
1. Computer systems—Evaluation. 2. Computer systems—Reliability. 3. Computer software—Validation. 4. Computer architecture—Evaluation. 5. System engineering. I. Title.
QA76.9.E94B66 2015
005.1′4—dc23
2014020070

ISBN-13: 978-0-321-83382-2
ISBN-10: 0-321-83382-1

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, August 2014

Executive Editor
Bernard Goodwin

Senior Development Editor
Chris Zahn

Managing Editor
John Fuller

Senior Project Editor
Kesel Wilson

Copy Editor
Barbara Wood

Indexer
Jack Lewis

Proofreader
Andrea Fox

Editorial Assistant
Michelle Housley

Cover Designer
Alan Clements

Compositor
LaurelTech

In memory of my father, Henry S. Bondi,
who liked eclectic solutions to problems,
and of my violin teacher, Fritz Rikko,
who taught me how to analyze and debug.

À tous qui ont attendu.

This page intentionally left blank

Contents

This page intentionally left blank

Preface

The performance engineering of computer systems and the systems they control concerns the methods, practices, and disciplines that may be used to ensure that the systems provide the performance that is expected of them. Performance engineering is a process that touches every aspect of the software lifecycle, from conception and requirements planning to testing and delivery. Failure to address performance concerns at the beginning of the software lifecycle significantly increases the risk that a software project will fail. Indeed, performance is the single largest risk to the success of any software project. Readers in the United States will recall that poor performance was the first sign that healthcare.gov, the federal web site for obtaining health insurance policies that went online in late 2013, was having a very poor start. News reports indicate that the processes and steps recommended in this book were not followed during its development and rollout. Performance requirements were inadequately specified, and there was almost no performance testing prior to the rollout because time was not available for it. This should be a warning that adequate planning and timely scheduling are preconditions for the successful incorporation of performance engineering into the software development lifecycle. "Building and then tuning" is an almost certain recipe for performance failure.

Scope and Purpose

The performance of a system is often characterized by the amount of time it takes to accomplish a variety of prescribed tasks and the number of times it can accomplish those tasks in a set time period. For example:

- A government system for selling health insurance policies to the general public, such as healthcare.gov, would be expected to determine an applicant's eligibility for coverage, display available options, and confirm the choice of policy and the premium due within designated amounts of time regardless of how many applications were to be processed within the peak hour.

- An online stock trading system might be expected to obtain a quote of the current value of a security within a second or so and execute a trade within an even shorter amount of time.
- A monitoring system, such as an alarm system, is expected to be able to process messages from a set of sensors and display corresponding status indications on a console within a short time of their arrival.
- A web-based news service would be expected to retrieve a story and display related photographs quickly.

This is a book about the practice of the performance engineering of software systems and software-controlled systems. It will help the reader address the following performance-related questions concerning the architecture, development, testing, and sizing of a computer system or a computer-controlled system:

- What capacity should the system have? How do you specify that capacity in both business-related and engineering terms?
- What business, social, and engineering needs will be satisfied by given levels of throughput and system response time?
- How many data records, abstract objects, or representations of concrete, tangible objects must the system be able to manage, monitor, and store?
- What metrics do you use to describe the performance your system needs and the performance it has?
- How do you specify the performance requirements of a system? Why do you need to specify them in the first place?
- How can the resource usage performance of a system be measured? How can you verify the accuracy of the measurements?
- How can you use mathematical models to predict a system's performance? Can the models be used to predict the performance if an application is added to the system or if the transaction rate increases?
- How can mathematical models of performance be used to plan performance tests and interpret the results?

- How can you test performance in a manner that tells you if the system is functioning properly at all load levels and if it will scale to the extent and in the dimensions necessary?
- What can poor performance tell you about how the system is functioning?
- How do you architect a system to be scalable? How do you specify the dimensions and extent of the scalability that will be required now or in the future? What architecture and design features undermine the scalability of a system?
- Are there common performance mistakes and misconceptions? How do you avoid them?
- How do you incorporate performance engineering into an agile development process?
- How do you tell the performance story to management?

Questions like these must be addressed at every phase of the software lifecycle. A system is unlikely to provide adequate performance with a cost-effective configuration unless its architecture is influenced by well-formulated, testable performance requirements. The requirements must be written in measurable, unambiguous, testable terms. Performance models may be used to predict the effects of design choices such as the use of scheduling rules and the deployment of functions on one or more hosts. Performance testing must be done to ensure that all system components are able to meet their respective performance needs, and to ensure that the end-to-end performance of the system meets user expectations, the owner's expectations, and, where applicable, industry and government regulations. Performance requirements must be written to help the architects identify the architectural and technological choices needed to ensure that performance needs are met. Performance requirements should also be used to determine how the performance of a system will be tested.

The need for performance engineering and general remarks about how it is practiced are presented in Chapter 1. Metrics are needed to describe performance quantitatively. A discussion of performance metrics is given in Chapter 2. Once performance metrics have been identified, basic analysis methods may be used to make predictions about system performance, as discussed in Chapter 3. The anticipated workload can be quantitatively described as in Chapter 4, and performance requirements can be specified. Necessary attributes of performance requirements and best practices for writing and managing them are discussed in Chapters 5 through 7. To understand the performance that has been attained and to verify that performance requirements have been met, the system must be measured. Techniques for doing so are given in Chapter 8. Performance tests should be structured to enable the evaluation of the scalability of a system, to determine its capacity and responsiveness, and to determine whether it is meeting throughput and response time requirements. It is essential to test the performance of all components of the system before they are integrated into a whole, and then to test system performance from end to end before the system is released. Methods for planning and executing performance tests are discussed in Chapter 9. In Chapter 10 we discuss procedures for evaluating the performance of a system and the practice of performance modeling with some examples. In Chapter 11 we discuss ways of describing system scalability and examine ways in which scalability is enhanced or undermined. Performance engineering pitfalls are examined in Chapter 12, and performance engineering in an agile context is discussed in Chapter 13. In Chapter 14 we consider ways of communicating the performance story. Chapter 15 contains a discussion of where to learn more about various aspects of performance engineering.

This book does not contain a presentation of the elements of probability and statistics and how they are applied to performance engineering. Nor does it go into detail about the mathematics underlying some of the main tools of performance engineering, such as queueing theory and queueing network models. There are several texts that do this very well already. Some examples of these are mentioned in Chapter 15, along with references on some detailed aspects of performance engineering, such as database design. Instead, this book focuses on various steps of the performance engineering process and the link between these steps and those of a typical software lifecycle. For example, the chapters on performance requirements engineering draw parallels with the engineering of functional requirements, and the chapter on scalability explains how performance models can be used to evaluate it and how architectural characteristics might affect it.

Audience

This book will be of interest to software and system architects, requirements engineers, designers and developers, performance testers, and product managers, as well as their managers. While all stakeholders should benefit from reading this book from cover to cover, the following stakeholders may wish to focus on different subsets of the book to begin with:

- Product owners and product managers who are reluctant to make commitments to numerical descriptions of workloads and requirements will benefit from the chapters on performance metrics, workload characterization, and performance requirements engineering.
- Functional testers who are new to performance testing may wish to read the chapters on performance metrics, performance measurement, performance testing, basic modeling, and performance requirements when planning the implementation of performance tests and testing tools.
- Architects and developers who are new to performance engineering could begin by reading the chapters on metrics, basic performance modeling, performance requirements engineering, and scalability.

This book may be used as a text in a senior- or graduate-level course on software performance engineering. It will give the students the opportunity to learn that computer performance evaluation involves integrating quantitative disciplines with many aspects of software engineering and the software lifecycle. These include understanding and being able to explain why performance is important to the system being built, the commercial and engineering implications of system performance, the architectural and software aspects of performance, the impact of performance requirements on the success of the system, and how the performance of the system will be tested.

This page intentionally left blank

Acknowledgments

This book is based in part on a training course entitled Foundations of Performance Engineering. I developed this course to train performance engineering and performance testing teams at various Siemens operating companies. The course may be taught on its own or, as my colleague Alberto Avritzer and I have done, as part of a consulting engagement. When teaching the course as part of a consulting engagement, one may have the opportunity to integrate the client's performance issues and even test data into the class material. This helps the clients resolve the particular issues they face and is effective at showing how the material on performance engineering presented here can be integrated into their software development processes.

One of my goals in writing this book was to relate this practical experience to basic performance modeling methods and to link performance engineering methods to the various stages of the software lifecycle. I was encouraged to write it by Dr. Dan Paulish, my first manager at Siemens Corporate Research (now Siemens Corporation, Corporate Technology, or SC CT); by Prof. Len Bass, who at the time was with the Software Engineering Institute in Pittsburgh; and by Prof. C. Murray Woodside of Carleton University in Ottawa. We felt that there was a teachable story to tell about the practical performance issues I have encountered during a career in performance engineering that began during the heyday of mainframe computers.

My thinking on performance requirements has been strongly influenced by Brian Berenbach, who has been a driving force in the practice of requirements engineering at SC CT. I would like to thank my former AT&T Labs colleagues, Dr. David Hoeflin and Dr. Richard Oppenheim, for reading and commenting on selected chapters. We worked together for many years as part of a large group of performance specialists. My experience in that group was inspiring and rewarding. I would also like to thank Dr. Alberto Avritzer of SC CT for many lively discussions on performance engineering.

I would like to thank the following past and present managers and staff at SC CT for their encouragement in the writing of this book.

Between them, Raj Varadarajan and Dr. Michael Golm read all of the chapters of the book and made useful comments before submission to the publisher.

Various Siemens operating units with whom I have worked on performance issues kindly allowed me to use material I had prepared for them in published work. Ruth Weitzenfeld, SC CT's librarian, cheerfully obtained copies of many references. Patti Schmidt, SC CT's in-house counsel, arranged for permission to quote from published work I had prepared while working at Siemens. Dr. Yoni Levi of AT&T Labs kindly arranged for me to obtain AT&T's permission to quote from a paper I had written on scalability while working there. This paper forms the basis for much of the content of Chapter 11.

I would like to thank my editors at Addison-Wesley, Bernard Goodwin and Chris Zahn, and their assistant, Michelle Housley, for their support in the preparation of this book. It has been a pleasure to work with them. The copy editor, Barbara Wood, highlighted several points that needed clarification. Finally, the perceptive comments of the publisher's reviewers, Nick Rozanski, Don Shafer, and Matthew Scarpino, have done much to make this a better book.

About the Author

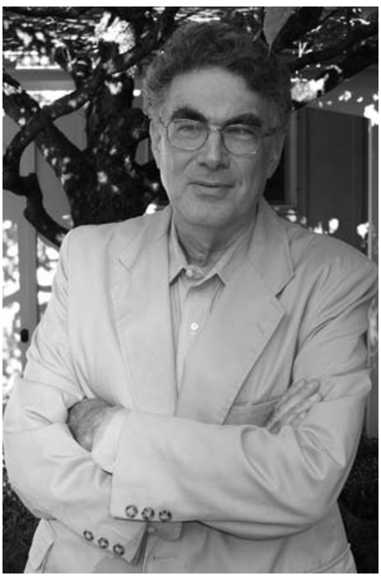


Photo by Rixt Bosma, www.rixtbosma.nl

André B. Bondi is a Senior Staff Engineer working in performance engineering at Siemens Corp., Corporate Technology, in Princeton, New Jersey. He has worked on performance issues in several domains, including telecommunications, conveyor systems, finance systems, building surveillance, railways, and network management systems. Prior to joining Siemens, he held senior performance positions at two start-up companies. Before that, he spent more than ten years working on a variety of performance and operational issues at AT&T Labs and its predecessor, Bell Labs. He has taught courses in performance, simulation, operating systems principles, and computer architecture at the University of California, Santa Barbara. Dr. Bondi holds a PhD in computer science from Purdue University, an MSc in statistics from University College London, and a BSc in mathematics from the University of Exeter.

This page intentionally left blank

Chapter 1

Why Performance Engineering? Why Performance Engineers?

This chapter describes the importance of performance engineering in a software project and explains the role of a performance engineer in ensuring that the system has good performance upon delivery. Overviews of different aspects of performance engineering are given.

1.1 Overview

The performance of a computer-based system is often characterized by its ability to perform defined sets of activities at fast rates and with quick response time. Quick response times, speed, and scalability are highly desired attributes of any computer-based system. They are also competitive differentiators. That is, they are attributes that distinguish a system from other systems with like functionality and make it more attractive to a prospective buyer or user.

If a system component has poor performance, the system as a whole may not be able to function as intended. If a system has poor performance, it will be unattractive to prospective users and buyers. If the project fails as a result, the investment in building the system will have been wasted. The foregoing is true whether the system is a command and control system, a transaction-based system, an information retrieval system, a video game, an entertainment system, a system for displaying news, or a system for streaming media.

The importance of performance may be seen in the following examples:

- A government-run platform for providing services on a grand scale must be able to handle a large volume of transactions from the date it is brought online. If it is not able to do so, it will be regarded as ineffective. In the United States, the federal web site for applying for health insurance mandated by the Affordable Care Act, healthcare.gov, was extremely slow for some time after it was made available to the public. According to press reports and testimony before the United States Congress, functional, capacity, and performance requirements were unclear. Moreover, the system was not subjected to rigorous performance tests before being brought online [Eilperin2013].
- An online securities trading system must be able to handle large numbers of transactions per second, especially in a volatile market with high trading volume. A brokerage house whose system cannot do this will lose business very quickly, because slow execution could lead to missing a valuable trading opportunity.
- An online banking system must display balances and statements rapidly. It must acknowledge transfers and the transmission of payments quickly for users to be confident that these transactions have taken place as desired.
- Regulations such as fire codes require that audible and visible alarms be triggered within 10 seconds of smoke being detected. In many jurisdictions, a building may not be used if the fire alarm system cannot meet this requirement.
- A telephone network must be able to handle large numbers of call setups and teardowns per second and provide such services as call forwarding and fraud detection within a short time of each call being initiated.

- A rail network control system must be able to monitor train movements and set signals and switches accordingly within very short amounts of time so that trains are routed to their correct destinations without colliding with one another.
- In combat, a system that has poor performance may endanger the lives or property of its users instead of endangering those of the enemy.
- A medical records system must be able to pull up patient records and images quickly so that retrieval will not take too much of a doctor's time away from diagnosis and treatment.

The foregoing examples show that performance is crucial to the correct functioning of a software system and of the application it controls. As such, performance is an attribute of system quality that presents significant business and engineering risks. In some applications, such as train control and fire alarm systems, it is also an essential ingredient of safety. Performance engineering mitigates these risks by ensuring that adequate attention is paid to them at every stage of the software lifecycle, while improving the capacity of systems, improving their response times, ensuring their scalability, and increasing user productivity. All of these are key competitive differentiators for any software product.

Despite the importance of system performance and the severe risk associated with inattentiveness to it, it is often ignored until very late in the software development cycle. Too often, the view is that performance objectives can be achieved by tuning the system once it is built. This mindset of "Build it, then tune it" is a recurring cause of the failure of a system to meet performance needs [SmithWilliams2001]. Most performance problems have their root causes in poor architectural choices. For example:

- An architectural choice could result in the creation of foci of overload.
- A decision is made that a set of operations that could be done in parallel on a multiprocessor or multicore host will be handled by a single thread. This would result in the onset of a software bottleneck for sufficiently large loads.

One of the possible consequences of detecting a performance issue with an architectural cause late in the software lifecycle is that a considerable amount of implementation work must be undone and redone.

This is needlessly expensive when one considers that the problem could have been avoided by performing an architectural review. This also holds for other quality attributes such as reliability, availability, and security.

1.2 The Role of Performance Requirements in Performance Engineering

To ensure that performance needs are met, it is important that they be clearly specified in requirements early in the software development cycle. Early and concise specifications of performance requirements are necessary because:

- Performance requirements are potential drivers of the system architecture and the choice of technologies to be used in the system's implementation. Moreover, many performance failures have their roots in poor architectural choices. Modification of the architecture before a system is implemented is cheaper than rebuilding a slow system from scratch.
- Performance requirements are closely related to the contractual expectations of system performance negotiated between buyer and seller, as well as to any relevant regulatory requirements such as those for fire alarm systems.
- The performance requirements will be reflected in the performance test plan.
- Drafting and reviewing performance requirements force the consideration of trade-offs between execution speed and system cost, as well as between execution speed and simplicity of both the architecture and the implementation. For instance, it is more difficult to design and correctly code a system that uses multithreading to achieve parallelism in execution than to build a single-threaded implementation.
- Development and/or hardware costs can be reduced if performance requirements that are found to be too stringent are relaxed early in the software lifecycle. For example, while a 1-second average response time requirement may be desirable, a 2-second requirement may be sufficient for business or engineering needs. Poorly specified performance

requirements can lead to confusion among stakeholders and the delivery of a poor-quality product with slow response times and inadequate capacity.

- If a performance issue that cannot be mapped to explicit performance requirements emerges during testing or production, stakeholders might not feel obliged to correct it.

We shall explore the principles of performance requirements in Chapter 5.

1.3 Examples of Issues Addressed by Performance Engineering Methods

Apart from mitigating business risk, performance engineering methods assist in answering a variety of questions about a software system. The performance engineer must frequently address questions related to capacity. For example:

- Can the system carry the peak load? The answer to this question depends on whether the system is adequately sized, and on whether its components can interact gracefully under load.
- Will the system cope with a surge in load and continue to function properly when the surge abates? This question is related to the reliability of the system. We do not want it to crash when it is most needed.
- What will be the performance impacts of adding new functionality to a system? To answer this question, we need to understand the extra work associated with each invocation of the functionality, and how often that functionality is invoked. We also need to consider whether the new functionality will adversely affect the performance of the system in its present form.
- Will the system be able to carry an increase in load? To answer this question, we must first ask whether there are enough resources to allow the system to perform at its current level.
- What is the performance impact of increasing the size of the user base? Answering this question entails understanding the memory and secondary storage footprints per user as well as in

total, and then being able to quantify the increased demand for memory, processing power, I/O, and network bandwidth.

- Can the system meet customer expectations or engineering needs if the average response time requirement is 2 seconds rather than 1 second? If so, it might be possible to build the system at a lower cost or with a simpler architecture. On the other hand, the choice of a simpler architecture could adversely affect the ability to scale up the offered load later, while still maintaining the response time requirement.
- Can the system provide the required performance with a cost-effective configuration? If it cannot, it will not fare well in the marketplace.

Performance can have an effect on the system's functionality, or its perceived functionality. If the system does not respond to an action before there is a timeout, it may be declared unresponsive or down if timeouts occur in a sufficiently large number of consecutive attempts at the action.

The performance measures of healthy systems tend to behave in a predictable manner. Deviations from predictable performance are signs of potential problems. Trends or wild oscillations in the performance measurements may indicate that the system is unstable or that a crash will shortly occur. For example, steadily increasing memory occupancy indicates a leak that could bring the system down, while oscillating CPU utilization and average response times may indicate that the system is repeatedly entering deadlock and timing out.

1.4 Business and Process Aspects of Performance Engineering

Ensuring the performance of a system entails initial and ongoing investment. The investment is amply rewarded by reductions in business risk, increased system stability, and system scalability. Because performance is often the single biggest risk to the success of a project [Bass2007], reducing this risk will make a major contribution to reducing the total risk to the project overall.

The initial performance engineering investments in a software project include

- Ensuring that there is performance engineering expertise on the project, perhaps including an individual designated as the lead performance engineer
- Drafting performance requirements
- Planning lab time for performance measurement and performance testing
- Acquiring and preparing performance measurement tools, load generation tools, and analysis and reporting tools to simplify the presentation and tracking of the results of the performance tests

Incorporating sound performance engineering practices into every aspect of the software development cycle can considerably reduce the performance risk inherent in the development of a large, complicated system. The performance process should be harmonized with the requirements, architectural, development, and testing phases of the development lifecycle. In addition to the steps just described, the performance engineering effort should include

1. A review of the system architecture from the standpoints of performance, reliability, and scalability
2. An evaluation of performance characteristics of the technologies proposed in the architecture specification, including quick performance testing of any proposed platforms [MBH2005]
3. Incremental performance testing following incremental functional testing of the system, followed by suggestions for architectural and design revisions as needed
4. Retesting to overcome the issues revealed and remedied as a result of the previous step

Performance engineering methods can also be used to manage cost-effective system growth and added functionality. For an existing system, growth is managed by building a baseline model based on measurements of resource usage and query or other work unit rates taken at runtime. The baseline model is combined with projected traffic rates to determine resource requirements using mathematical models and other methods drawn from the field of operations research [LZGS1984, Kleinrock1975, Kleinrock1976, MenasceAlmeida2000].

We now turn to a discussion of the various disciplines and techniques a performance engineer can use to perform his or her craft.

1.5 Disciplines and Techniques Used in Performance Engineering

The practice of performance engineering draws on many disciplines and skills, ranging from the technological to the mathematical and even the political. Negotiating, listening, and writing skills are also essential for successful performance engineering, as is the case for successful architects and product owners. The set of original undergraduate major subjects taken by performance engineers the author has met includes such quantitative disciplines as mathematics, physics, chemical engineering, chemistry, biology, electrical engineering, statistics, economics, and operations research, as well as computer science. Those who have not majored in computer science will need to learn about such subjects as operating systems design, networking, and hardware architecture, while the computer scientists may need to acquire additional experience with working in a quantitative discipline.

To understand resource usage and information flow, the performance engineer must have at least a rudimentary knowledge of computer systems architecture, operating systems principles, concurrent programming principles, and software platforms such as web servers and database management systems. In addition, the performance engineer must have a sound grasp of the technologies and techniques used to measure resource usage and traffic demands, as well as those used to drive transactions through a system under test.

To understand performance requirements and the way the system will be used, it is necessary to know something about its domain of application. The performance and reliability needs of financial transaction systems, fire alarm systems, network management systems, conveyor belts, telecommunications systems, train control systems, online news services, search engines, and multimedia streaming services differ dramatically. For instance, the performance of fire alarm systems is governed by building and fire codes in the jurisdictions where the systems will be installed, while that of a telephone system may be governed by international standards. The performance needs of all the systems mentioned previously may be driven by commercial considerations such as competitive differentiation.

Because performance is heavily influenced by congestion, it is essential that a performance engineer be comfortable with quantitative analysis methods and have a solid grasp of basic statistics, queueing theory, and simulation methods. The wide variety of computer technologies and the evolving set of problem domains mean that the performance engineer should have an eclectic set of skills and analysis methods at his or her disposal. In addition, it is useful for the performance engineer to know how to analyze large amounts of data with tools such as spreadsheets and scripting languages, because measurement data from a wide variety of sources may be encountered. Knowledge of statistical methods is useful for planning experiments and for understanding the limits of inferences that can be drawn from measurement data. Knowledge of queueing theory is useful for examining the limitations of design choices and the potential improvements that might be gained by changing them.

While elementary queueing theory may be used to identify limits on system capacity and to predict transaction loads at which response times will suddenly increase [DenningBuzen1978], more complex queueing theory may be required to examine the effects of service time variability, interarrival time variability, and various scheduling rules such as time slicing, preemptive priority, nonpreemptive priority, and cyclic service [Kleinrock1975, Kleinrock1976].

Complicated scheduling rules, load balancing heuristics, protocols, and other aspects of system design that are not susceptible to queueing analysis may be examined using approximate queueing models and/or discrete event simulations, whose outputs should be subjected to statistical analysis [LawKelton1982].

Queueing models can also be used in sizing tools to predict system performance and capacity under a variety of load scenarios, thus facilitating what-if analysis. This has been done with considerable commercial success. Also, queueing theory can be used to determine the maximum load to which a system should be subjected during performance tests once data from initial load test runs is available.

The performance engineer should have some grasp of computer science, software engineering, software development techniques, and programming so that he or she can quickly recognize the root causes of performance issues and negotiate design trade-offs between architects and developers when proposing remedies. A knowledge of hardware architectures, including processors, memory architectures, network technologies, and secondary storage technologies, and the ability to learn about new technologies as they emerge are very helpful to the performance engineer as well.

Finally, the performance engineer will be working with a wide variety of stakeholders. Interactions will be much more fruitful if the performance engineer is acquainted with the requirements drafting and review processes, change management processes, architecture and design processes, and testing processes. The performance engineer should be prepared to work with product managers and business managers. He or she will need to explain choices and recommendations in terms that are related to the domain of application and to the trade-offs between costs and benefits.

1.6 Performance Modeling, Measurement, and Testing

Performance modeling can be used to predict the performance of a system at various times during its lifecycle. It can be used to characterize capacity; to help understand the impact of proposed changes, such as changes to scheduling rules, deployment scenarios, technologies, and traffic characteristics; or to predict the effect of adding or removing workloads. Deviations from the qualitative behavior predicted by queueing models, such as slowly increasing response times or memory occupancy when the system load is constant or expected to be constant, can be regarded as indications of anomalous system behavior. Performance engineers have used their understanding of performance models to identify software flaws; software bottlenecks, especially those occurring in new technologies that may not yet be well understood [ReeserHariharan2000]; system malfunctions (including the occurrence of deadlocks); traffic surges; and security violations. This has been done by examining performance measurement data, the results of simulations, and/or queueing models [AvBonWey2005, AvTanJaCoWey2010]. Interestingly, the principles that were used to gain insights into performance in these cases were independent of the technologies used in the system under study.

Performance models and statistical techniques for designing experiments can also be used to help us plan and interpret the results of performance tests.

An understanding of rudimentary queueing models will help us determine whether the measurement instrumentation is yielding valid values of performance metrics.

Pilot performance tests can be used to identify the ranges of transaction rates for which the system is likely to be lightly, moderately, or heavily loaded. Performance trends with respect to load are useful for predicting capacity and scalability. Performance tests at loads near or above that at which any system resource is likely to be saturated will be of no value for predicting scalability or performance, though they can tell us whether the system is likely to crash when saturated, or whether the system will recover gracefully once the load is withdrawn. An understanding of rudimentary performance models will help us to design performance tests accordingly.

Methodical planning of experiments entails the identification of factors to be varied from one test run to the next. Fractional replication methods help the performance engineer to choose telling subsets of all possible combinations of parameter settings to minimize the number of experiments that must be done to predict performance.

Finally, the measurements obtained from performance tests can be used as the input parameters of sizing tools (based on performance models) that will assist in sizing and choosing the configurations needed to carry the anticipated load to meet performance requirements in a cost-effective manner.

1.7 Roles and Activities of a Performance Engineer

Like a systems architect, a performance engineer should be engaged in all stages of a software project. The performance engineer is frequently a liaison between various groups of stakeholders, including architects, designers, developers, testers, product management, product owners, quality engineers, domain experts, and users. The reasons for this are:

- The performance of a system affects its interaction with the domain.
- Performance is influenced by every aspect of information flow, including
  - The interactions between system components
  - The interactions between hardware elements and domain elements
  - The interactions between the user interface and all other parts of the system
  - The interactions between component interfaces

When performance and functional requirements are formulated, the performance engineer must ensure that performance and scalability requirements are written in verifiable, measurable terms, and that they are linked to business and engineering needs. At the architectural stage, the performance engineer advises on the impacts of technology and design choices on performance and identifies impediments to smooth information flow. During design and development, the performance engineer should be available to advise on the performance characteristics and consequences of design choices and scheduling rules, indexing structures, query patterns, interactions between threads or between devices, and so on. During functional testing, including unit testing, the performance engineer should be alerted if the testers feel that the system is too slow. This can indicate a future performance problem, but it can also indicate that the system was not configured properly. For example, a misconfigured IP address could result in an indication by the protocol implementation that the targeted host is unresponsive or nonexistent, or in a failure of one part of the system to connect with another. It is not unusual for the performance engineer to be involved in diagnosing the causes of these problems, as well as problems that might appear in production.

The performance engineer should be closely involved in the planning and execution of performance tests and the interpretation of the results. He or she should also ensure that the performance instrumentation is collecting valid measurement data and generating valid loads. Moreover, it is the performance engineer who supervises the preparation of reports of performance tests and measurements in production, explains them to stakeholders, and mediates negotiations between stakeholders about necessary and possible modifications to improve performance.

If the performance of a system is found to be inadequate, whether in testing or in production, the performance engineer will be able to play a major role in diagnosing the technical cause of the problem. Using the measurement and testing methods described in this book, the performance engineer works with testers and architects to identify the nature of the cause of the problem and with developers to determine the most cost-effective way to fix it. Historically, the performance engineer's first contact with a system has often been in "repairman mode" when system performance has reached a crisis point. It is preferable that performance issues be anticipated and avoided during the early stages of the software lifecycle.

The foregoing illustrates that the performance engineer is a performance advocate and conscience for the project, ensuring that performance needs are anticipated and accounted for at every stage of the development cycle, the earlier the better [Browne1981]. Performance advocacy includes the preparation of clear summaries of performance status, making recommendations for changes, reporting on performance tests, and reporting on performance issues in production. Thus, the performance engineer should not be shy about blowing the whistle if a major performance problem is uncovered or anticipated. The performance reports should be concise, cogent, and pungent, because stakeholders such as managers, developers, architects, and product owners have little time to understand the message being communicated. Moreover, the performance engineer must ensure that graphs and tables tell a vivid and accurate story.

In the author's experience, many stakeholders have little training or experience in quantitative methods unless they have worked in disciplines such as statistics, physics, chemistry, or econometrics before joining the computing profession. Moreover, computer science and technology curricula seldom require the completion of courses related to performance evaluation for graduation. This means that the performance engineer must frequently play the role of performance teacher while explaining performance considerations in terms that can be understood by those trained in other disciplines.

1.8 Interactions and Dependencies between Performance Engineering and Other Activities

Performance engineering is an iterative process involving interactions between multiple sets of stakeholders at many stages of the software lifecycle (see Figure 1.1). The functional requirements inform the specification of the performance requirements. Both influence the architecture and the choice of technology. Performance requirements may be formulated with the help of performance models. The models are used to plan performance tests to verify scalability and that performance requirements have been met. Performance models may also be used in the design of modifications. Data gathered through performance monitoring and capacity planning may be used to determine whether new functionality or load may be added to the system.

The performance engineer must frequently take responsibility for ensuring that these interactions take place. None of the activities and skills we have mentioned is sufficient for the practice of performance engineering in and of itself.

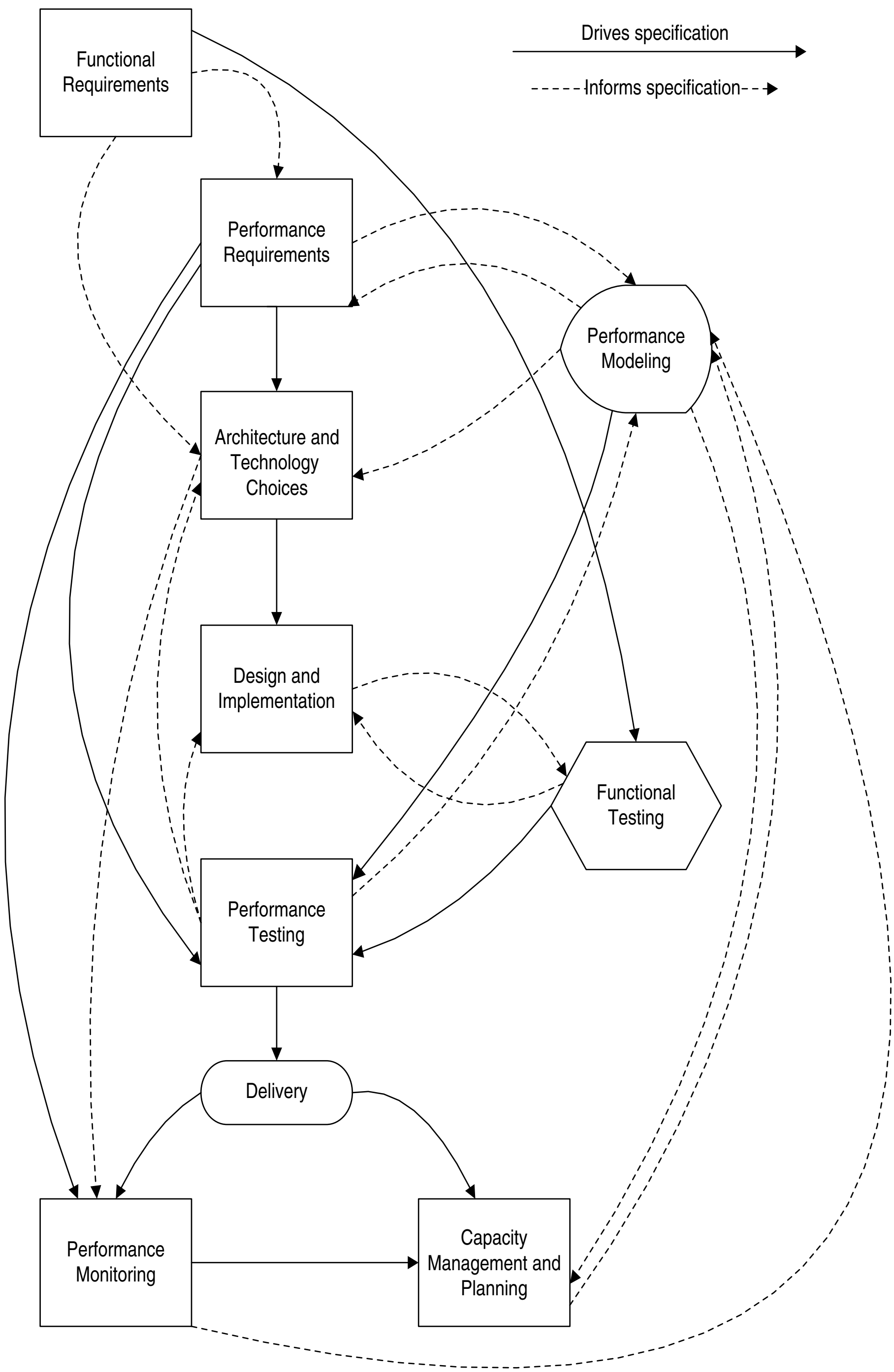


Figure 1.1 *Interactions between performance engineering activities and other software lifecycle activities*

1.9 A Road Map through the Book

Performance metrics are described in Chapter 2. One needs performance metrics to be able to define the desired performance characteristics of a system, and to describe the characteristics of the performance of an existing system. In the absence of metrics, the performance requirements of a system can be discussed only in vague terms, and the requirements cannot be specified, tested, or enforced.

Basic performance modeling and analysis are discussed in Chapter 3. We show how to establish upper bounds on system throughput and lower bounds on system response time given the amount of time it takes to do processing and I/O. We also show how rudimentary queueing models can be used to make predictions about system response time when a workload has the system to itself and when it is sharing the system with other workloads.

In Chapter 4 we explore methods of characterizing the workload of a system. We explain that workload characterization involves understanding what the system does, how often it is required to do it, why it is required to do it, and the performance implications of the nature of the domain of application and of variation in the workload over time.

Once the workload of the system has been identified and understood, we are in a position to identify performance requirements. The correct formulation of performance requirements is crucial to the choice of a sound, cost-effective architecture for the desired system. In Chapter 5 we describe the necessary attributes of performance requirements, including linkage to business and engineering needs, traceability, clarity, and the need to express requirements unambiguously in terms that are measurable, testable, and verifiable. These are preconditions for enforcement. Since performance requirements may be spelled out in contracts between a buyer and a supplier, enforceability is essential. If the quantities specified in a performance requirement cannot be measured, the requirement is deficient and unenforceable and should either be flagged as such or omitted. In Chapter 6 we discuss specific types of the ability of a system to sustain a given load, the metrics used to describe performance requirements, and performance requirements related to networking and to specific domains of application. In Chapter 7 we go into detail about how to express performance requirements clearly and how they can be managed.

One must be able to measure a system to see how it is functioning, to identify hardware and software bottlenecks, and to determine whether it is meeting performance requirements. In Chapter 8 we describe performance measurement tools and instrumentation that can help one do this. Instrumentation that is native to the operating system measures resource usage (e.g., processor utilization and memory usage) and packet traffic through network ports. Tools are available to measure activity and resource usage of particular system components such as databases and web application servers. Application-level measurements and load drivers can be used to measure system response times. We also discuss measurement pitfalls, the identification of incorrect measurements, and procedures for conducting experiments in a manner that helps us learn about system performance in the most effective way.

Performance testing is discussed in Chapter 9. We show how performance test planning is linked to both performance requirements and performance modeling. We show how elementary performance modeling methods can be used to interpret performance test results and to identify system problems if the tests are suitably structured. Among the problems that can be identified are concurrent programming bugs, memory leaks, and software bottlenecks. We discuss suitable practices for the documentation of performance test plans and results, and for the organization of performance test data.

In Chapter 10 we use examples to illustrate the progression from system understanding to model formulation and validation. We look at cases in which the assumptions underlying a conventional performance model might deviate from the properties of the system of interest. We also look at the phases of a performance modeling study, from model formulation to validation and performance prediction.

Scalability is a desirable attribute of systems that is frequently mentioned in requirements without being defined. In the absence of definitions, the term is nothing but a buzzword that will engender confusion at best. In Chapter 11 we look in detail at ways of characterizing the scalability of a system in different dimensions, for instance, in terms of its ability to handle increased loads, called *load scalability*, or in terms of the ease or otherwise of expanding its structure, called *structural scalability*. In this chapter we also provide examples of cases in which scalability breaks down and discuss how it can be supported.

Intuition does not always lead to correct performance engineering decisions, because it may be based on misconceptions about what scheduling algorithms or the addition of multiple processors might contribute to system performance. This is the reason Chapter 12, which contains a discussion of performance engineering pitfalls, appears in this book. In this chapter we will learn that priority scheduling does not increase the processing capacity of a system. It can only reduce the response times of jobs that are given higher priority than others and hence reduce the times that these jobs hold resources. Doubling the number of processors need not double processing capacity, because of increased contention for the shared memory bus, the lock for the run queue, and other system resources. In Chapter 12 we also explore pitfalls in system measurement, performance requirements engineering, and other performance-related topics.

The use of agile development processes in performance engineering is discussed in Chapter 13. We will explore how agile methods might be used to develop a performance testing environment even if agile methods have not been used in the development of the system as a whole. We will also learn that performance engineering as part of an agile process requires careful advance planning and the implementation of testing tools. This is because the time constraints imposed by short sprints necessitate the ready availability of load drivers, measurement tools, and data reduction tools.

In Chapter 14 we explore ways of learning, influencing, and telling the performance story to different sets of stakeholders, including architects, product managers, business executives, and developers.

Finally, in Chapter 15 we point the reader to sources where more can be learned about performance engineering and its evolution in response to changing technologies.

1.10 Summary

Good performance is crucial to the success of a software system or a system controlled by software. Poor performance can doom a system to failure in the marketplace and, in the case of safety-related systems, endanger life, the environment, or property. Performance engineering practice contributes substantially to ensuring the performance of a product and hence to the mitigation of the business risks associated with software performance, especially when undertaken from the earliest stages of the software lifecycle.

This page intentionally left blank

Index

C

R

This page intentionally left blank