Software Performance
and Scalability
Press Operating Committee

Chair
Linda Shafer
former Director, Software Quality Institute
The University of Texas at Austin

Editor-in-Chief
Alan Clements
Professor
University of Teesside

Board Members
David Anderson, Principal Lecturer, University of Portsmouth
Mark J. Christensen, Independent Consultant
James Conrad, Associate Professor, UNC Charlotte
Michael G. Hinchey, Director, Software Engineering Laboratory, NASA Goddard Space Flight Center
Phillip Laplante, Associate Professor, Software Engineering, Penn State University
Richard Thayer, Professor Emeritus, California State University, Sacramento
Donald F. Shafer, Chief Technology Officer, Athens Group, Inc.
Janet Wilson, Product Manager, CS Press

IEEE Computer Society Publications
The world-renowned IEEE Computer Society publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. These books are available from most retail outlets. Visit the CS Store at http://computer.org/cspress for a list of products.

IEEE Computer Society / Wiley Partnership
The IEEE Computer Society and Wiley partnership allows the CS Press authored book program to produce a number of exciting new titles in areas of computer science, computing and networking with a special focus on software engineering. IEEE Computer Society members continue to receive a 15% discount on these titles when purchased through Wiley or at wiley.com/ieee.

To submit questions about the program or send proposals e-mail j.wilson@computer.org.
Telephone +1-714-821-8380.
Additional information regarding the Computer Society authored book program can also be accessed from our web site at http://computer.org/cspress.
Software Performance and Scalability
A Quantitative Approach

Henry H. Liu
To my family
Contents

PREFACE xv

ACKNOWLEDGMENTS xxi

Introduction 1
Performance versus Scalability / 1

PART 1 THE BASICS 3

1. Hardware Platform 5
   1.1 Turing Machine / 6
   1.2 von Neumann Machine / 7
   1.3 Zuse Machine / 8
   1.4 Intel Machine / 9
      1.4.1 History of Intel’s Chips / 9
      1.4.2 Hyperthreading / 9
      1.4.3 Intel’s Multicore Microarchitecture / 13
      1.4.4 Challenges for System Monitoring Tools / 17
   1.5 Sun Machine / 17
   1.6 System Under Test / 18
      1.6.1 Processors / 18
      1.6.2 Motherboard / 19
      1.6.3 Chipset / 20
# Contents

1.6.4 Storage / 22  
1.6.5 RAID / 24  
1.6.6 Networking / 27  
1.6.7 Operating System / 29  
1.7 Odds Against Turing / 30  
1.7.1 Memory Leaks / 30  
1.7.2 SLAs / 35  
1.8 Sizing Hardware / 35  
1.9 Summary / 37  
Recommended Reading / 37  
Exercises / 38  

2. Software Platform  
2.1 Software Stack / 42  
2.2 APIs / 44  
2.2.1 Windows APIs / 45  
2.2.2 Java APIs / 45  
2.2.3 Google APIs / 46  
2.3 Multithreading / 47  
2.4 Categorizing Software / 53  
2.4.1 Systems Software / 53  
2.4.2 Application Software / 54  
2.4.3 Middleware Software / 55  
2.5 Enterprise Computing / 55  
2.5.1 What Is Enterprise Software? / 55  
2.5.2 Enterprise Software Architecture / 57  
2.5.3 Monolithic Architecture / 57  
2.5.4 Client/Server Architecture / 58  
2.5.5 Three-Tier Architecture / 59  
2.5.6 N-Tier Architecture / 60  
2.5.7 Software Componentry / 61  
2.5.8 Service-Oriented Architecture / 61  
2.6 Summary / 63  
Recommended Reading / 64  
Exercises / 64  

3. Testing Software Performance and Scalability  
3.1 Scope of Software Performance and Scalability Testing / 67  
3.1.1 Performance Regression Testing / 68  
3.1.2 Performance Optimization and Tuning Testing / 70  
3.1.3 Performance Benchmarking Testing / 75
### 4. Case Study I: Queuing Theory Applied to SOA

5.1 Introduction to SOA / 178
5.2 XML Web Services / 179
5.3 The Analytical Model / 181
5.4 Service Demand / 183
   5.4.1 Web Services Handle Creation / 184
   5.4.2 XML SOAP Serialization/Deserialization / 184
   5.4.3 Network Latency / 185
   5.4.4 XML Web Service Provider / 186
   5.4.5 Database Server / 186
   5.4.6 Data Storage / 187
5.5 MedRec Application / 188
   5.5.1 Exposing a Stateless Session EJB as an XML Web Service / 188
   5.5.2 Consuming an XML Web Service Using SOAP / 189
5.6 MedRec Deployment and Test Scenario / 189
5.7 Test Results / 191
   5.7.1 Overhead of the XML Web Services Handle / 192
   5.7.2 Effects of Caching Web Services Handle / 193
   5.7.3 Throughput Dynamics / 194
   5.7.4 Bottleneck Analysis / 195

5.8 Comparing the Model with the Measurements / 198
5.9 Validity of the SOA Performance Model / 200
5.10 Summary / 200
Recommended Reading / 201
Exercises / 202

6. Case Study II: Queuing Theory Applied to Optimizing and Tuning Software Performance and Scalability / 205

6.1 Analyzing Software Performance and Scalability / 207
   6.1.1 Characterizing Performance and Scalability Problems / 207
   6.1.2 Isolating Performance and Scalability Factors / 208
   6.1.3 Applying Optimization and Tuning / 215

6.2 Effective Optimization and Tuning Techniques / 220
   6.2.1 Wait Events and Service Demands / 221
   6.2.2 Array Processing—Reducing \( V_i \) / 223
   6.2.3 Caching—Reducing Wait Time \( (W_i) \) / 226
   6.2.4 Covering Index—Reducing Service Demand \( (D_i) \) / 228
   6.2.5 Cursor-Sharing—Reducing Service Demand \( (D_i) \) / 229
   6.2.6 Eliminating Extraneous Logic—Reducing Service Demand \( (D_i) \) / 231
   6.2.7 Faster Storage—Reducing Data Latency \( (W_i) \) / 232
   6.2.8 MPLS—Reducing Network Latency \( (W_i) \) / 233
   6.2.9 Database Double Buffering—An Anti Performance and Scalability Pattern / 235

6.3 Balanced Queuing System / 240
6.4 Summary / 244
Recommended Reading / 245
Exercises / 246

PART 3 APPLYING API PROFILING / 249

7. Defining API Profiling Framework / 251
   7.1 Defense Lines Against Software Performance and Scalability Defects / 252
   7.2 Software Program Execution Stack / 253
   7.3 The PerfBasic API Profiling Framework / 254
10. Case Study: Applying API Profiling to Solving Software Performance and Scalability Challenges 303
10.1 Enabling API Profiling / 304
   10.1.1 Mechanism of Populating Log Entry / 305
   10.1.2 Source and Target Projects / 306
   10.1.3 Setting apf.properties File / 306
   10.1.4 Parsing Workflow / 308
   10.1.5 Verifying the Profiling-Enabled Source Code / 310
   10.1.6 Recommended Best Coding Practices / 311
   10.1.7 Enabling Non-Java Programs / 312
10.2 API Profiling with Standard Logs / 313
   10.2.1 Generating API Profiling Log Data / 313
   10.2.2 Parsing API Profiling Log Data / 314
   10.2.3 Generating Performance Maps / 316
   10.2.4 Making Sense Out of Performance Maps / 319
10.3 API Profiling with Custom Logs / 320
   10.3.1 Using Adapter to Transform Custom Logs / 320
   10.3.2 Generating Performance Maps with Custom Logs / 321
10.4 API Profiling with Combo Logs / 325
   10.4.1 Client Side Performance Map / 325
   10.4.2 Server Side Performance Map / 327
10.5 Applying API Profiling to Solving Performance and Scalability Problems / 333
   10.5.1 Baseline / 333
   10.5.2 Optimization / 335
   10.5.3 Analysis / 336
10.6 Summary / 337
Exercises / 338

APPENDIX A STOCHASTIC EQUILIBRIUM AND ERGODICITY 339
A.1 Basic Concepts / 339
   A.1.1 Random Variables / 339
   A.1.2 Random Variable Vector / 340
   A.1.3 Independent and Identical Distributions (IID) / 341
   A.1.4 Stationary Processes / 342
   A.1.5 Processes with Stationary Independent Increments / 342
A.2 Classification of Random Processes / 343
  A.2.1 General Renewal Processes / 343
  A.2.2 Markov Renewal Processes / 343
  A.2.3 Markov Processes / 343
A.3 Discrete-Time Markov Chains / 345
  A.3.1 Transition Probability Matrix and C-K Equations / 345
  A.3.2 State Probability Matrix / 347
  A.3.3 Classification of States and Chains / 348
A.4 Continuous-Time Markov Chains / 349
  A.4.1 C–K Equations / 349
  A.4.2 Transition Rate Matrix / 349
  A.4.3 Imbedded Markov Chains / 350
A.5 Stochastic Equilibrium and Ergodicity / 351
  A.5.1 Definition / 351
  A.5.2 Limiting State Probabilities / 353
  A.5.3 Stationary Equations / 354
  A.5.4 Ergodic Theorems for Discrete-Time Markov Chains / 354
  A.5.5 Ergodic Theorems for Continuous-Time Markov Chains / 356
A.6 Birth–Death Chains / 357
  A.6.1 Transition Rate Matrix / 357
  A.6.2 C–K Equations / 358
  A.6.3 Limiting State Probabilities / 359
  A.6.4 Ergodicity / 359

APPENDIX B  MEMORYLESS PROPERTY OF THE EXPONENTIAL DISTRIBUTION 361

APPENDIX C  M/M/1 QUEUES AT STEADY STATE 363
  C.1 Review of Birth–Death Chains / 363
  C.2 Utilization and Throughput / 364
  C.3 Average Queue Length in the System / 365
  C.4 Average System Time / 365
  C.5 Average Wait Time / 366

INDEX 367
Preface

Software platforms are a written product of the mind.
—D. S. Evans, A. Hagiu, and R. Schmalensee

WHY THIS BOOK

Few people would disagree with the fact that building a large-scale, high-performance, and scalable software system is a complex task. This is evidenced by the magnitude of required up-front and ongoing financial costs and personnel commonly seen at every large software development organization. Seeking effective, efficient, and economical approaches to developing large-scale software is of interest to the entire software community.

Regardless of its complexity and scope, every software development project is driven by a few common factors:

- It is required to be on schedule because of urgency to be first to market in order to gain a competitive edge.
- It is required to be within budget under the pressure of showing profit and return on investment (ROI) as soon as possible.
- It is required to provide customers with all major functionalities at a minimum.
- And it is required to meet customer’s expectations on performance and scalability to be usable.

While management is responsible for providing sufficient budget to cover personnel, development, test infrastructure, and so on, we, the technical staff (developers, quality assurance engineers, and performance engineers), are accountable for delivering the software product under development on schedule and within budget while meeting high standards on performance and scalability.
However, it’s not uncommon to see that performance and scalability are pushed aside by the following higher priority activities:

- Analyzing system functionality requirements
- Deciding on the right architecture and design patterns
- Choosing appropriate programming paradigms and efficient development tools
- Starting coding and delivering early builds that meet major functionality requirements as soon as possible
- Implementing automated functionality test frameworks

Performance and scalability are often an afterthought during the last minute of product release. And even worse, performance and scalability issues might actually be raised by unsatisfied customers as soon as the product is rushed to the market. Under such circumstances, intense pressure builds up internally, and panic and fire-fighting-like chaos ensues.

On the other hand, software performance and scalability are indeed very challenging technical issues. Precautions must be taken with every major effort to improve the performance and scalability of a software product. A few industrial pioneers have issued warnings:

- “More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity.” —W. A. Wulf
- “Premature optimization is the root of all evil.” —Tony Hoare and Donald Knuth
- “Bottlenecks occur in surprising places, so don’t try to second guess and put in a speed hack until you have proven that’s where the bottleneck is.” —Rob Pike

So, how can we implement an effective, efficient, and economical approach to building performance and scalability into software? Establishing a very capable performance and scalability test team would certainly help. However, it is my observation that this approach is insufficient for guaranteeing that performance and scalability issues are dealt with properly, as it may easily exclude software developers from taking performance and scalability concerns into account in the first place. It’s a reactive and less efficient approach to let the performance and scalability test engineers find the performance and scalability defects and then fix them with the developers. It’s a lot more costly to fix software performance and scalability defects without having the developers take care of them in the first place.

That’s the motivation behind this book, which promotes a proactive approach of letting the software developers build the performance and scalability into the product and letting the performance and scalability test engineers concentrate on the performance and scalability verification tests with a larger volume of data, more representative workloads, and more powerful hardware. This approach requires a mindset shift for the software developers that their job is just to make the software work and
the performance and scalability problems can be fixed outside their job scope. Software developers should think consciously from performance and scalability perspectives whenever they make a design or an implementation decision.

Software developers already possess strong, valuable, and hard-to-obtain software design and implementation skills. Regardless of their experience, they can complement their existing coding skills by acquiring from this book knowledge about designing and implementing performance and scalability into their products in the first place during the various life cycles of development.

Of course, it’s impractical to have only the software developers take care of all the performance and scalability challenges. Building a software system that performs and scales is a cross-team effort. This book provides a common knowledge platform for all stakeholders to work together to tame the challenging performance and scalability issues so that the product they are all responsible for is built to perform and scale.

**WHO THIS BOOK IS FOR**

If you are reading this book, probably you are interested in learning how you can help design and build performance and scalability into your software for which you are one of the stakeholders, either from the technical or management perspective. No matter what your roles are, I am very confident that you will learn something from this book that can help you become more knowledgeable, more productive, and more efficient in solving your performance and scalability issues.

I wrote this book with some specific groups of readers in my mind. In deciding what material to include in this book and how to write it, I tried my best to make this book pertinent and useful for the following groups of readers:

- **Software developers who have the most influence on how well the software product they develop will actually perform and scale.** If software developers are equipped with adequate knowledge and experience in software performance and scalability, fewer defects will slip out of their hands into the builds they deliver.

- **Software engineers who conduct the performance and scalability tests to make sure that the product will not be released to the market without catching and resolving major performance and scalability defects.** Nowadays, it’s very hard to find experienced software performance engineers. Most of the engineers who conduct the performance and scalability tests are from other job responsibilities, such as quality assurance, system administration, database administration, or programming. This book can help them get up to speed quickly in helping resolve software performance and scalability issues they discover through their tests.

- **Software performance managers and development managers who are interested in understanding software performance and scalability problems at a high level so that they can lead more effectively in getting various performance and scalability defects resolved in time.**
The book can also be used as a textbook in various ways. First of all, it can be used as a textbook for university courses related to computer performance evaluation and software non-functional testing at the upper-division undergraduate and graduate levels. It can be used as a required supplement to the computer organization texts now in use that every CS and CpE student must take. It is an ideal text as well to supplement a course in queuing theory that is available in many universities for the students majoring in mathematics, probability and statistics.

Many books available today on the subject of software performance and scalability do not provide the same level of quantitativeness, which is one of the most distinctive merits of this book. In my opinion, quantitativeness is a requirement for dealing with software performance and scalability issues, as performance and scalability are quantitatively measurable attributes of a software system.

I hope that the quantitative approach and the real-world quantitative case studies presented throughout this book can help you learn about software performance and scalability faster and more effectively. And more importantly, I am confident that by applying everything you learn from this book to your product, you can make a huge difference in improving the performance and scalability of your product to the satisfaction of your company and customers.

HOW THIS BOOK IS ORGANIZED

Software Performance and Scalability: A Quantitative Approach is the first book to focus on software performance and scalability in a quantitative approach. It introduces the basic concepts and principles behind the physics of software performance and scalability from a practical point of view. It demonstrates how the performance and scalability of your products can be optimized and tuned using both proven theories and quantitative, real-world examples presented as case studies in each chapter. These case studies can easily be applied to your software projects so that you can realize immediate, measurable improvements on the performance and scalability of your products.

As illustrated in Figure A, this book elaborates on three levels of skill sets for coping with software performance and scalability problems.

Figure A  Three levels of skill sets for solving software performance and scalability challenges.
Specifically, this book consists of the following three parts:

- **Part 1: The Basics.** This part lays the foundation for understanding the factors that affect the performance and scalability of a software product in general. It introduces the various hardware components of a modern computer system as well as software platforms that predetermine the performance and scalability of a software product. It concludes with how to test quantitatively the performance and scalability of a software product. Through quantitative measurements, you can determine not only which hardware and software platforms can deliver the required performance and scalability for your products, but also how to optimize and tune the performance and scalability of your products over time.

- **Part 2: Applying Queuing Theory.** Queuing theory is the mathematical study of waiting lines or queues for a system that depends on limited resources to complete certain tasks. It is particularly useful as a quantitative framework to help identify the performance and scalability bottlenecks of a computer software system. The efficacy of queuing theory in solving software performance and scalability problems is demonstrated in two subsequent chapters using quantitative case studies.

- **Part 3: Applying API Profiling.** API profiling provides quantitative information about how a software program is executed internally at the API level. Such information is useful in identifying the most expensive execution paths from performance and scalability perspectives. Based on such information, developers can design more efficient algorithms and implementations to achieve the best possible performance and scalability for products. This part introduces a generic API profiling framework (perfBasic), which can be implemented easily in any high-level programming languages. It concludes with a case study chapter showing quantitatively how one can use the performance maps generated with the API profiling data out of this API profiling framework to help solve software performance and scalability issues.

In order to make this book more suitable as a textbook for an upper division undergraduate or graduate level course for computer and software engineering students, exercises have been provided at the end of each chapter. In most cases, the exercises have been designed to encourage the reader to conduct his/her own research and come up with the quantitative solutions to the exercises. In addition, the reader is encouraged to think and practice, rather than simply writing a program or filling in a formula with numbers. Dealing with software performance and scalability problems is more challenging than simply coding, and oftentimes, it’s really passion and discipline that can make a difference.

I have made every effort to make this book concise, practical, interesting, and useful for helping you solve your software performance and scalability problems. I hope you’ll enjoy reading this book, apply what you learn from this book to your work, and see immediate positive results. In addition, be conscious that by developing high-performance and scalable software that consumes less electric power to run,
you are not only contributing to the success of your company and your customers, but also helping reduce global warming effects, for which we are all responsible.

HOW TO REACH THE AUTHOR

All mistakes and errors, if any, in the text are my responsibility. You are more than welcome to email me your comments about the contents of this book, or errors found therein, at henry@perfmath.com For any downloads and updated information about the book, visit the book’s website at http://www.perfmath.com.

HENRY H. LIU, PhD

Folsom, California
September 2008
Acknowledgments

First, I would like to thank all of the colleagues I had worked with in the field of physics research. Some of the greatest physicists that I was so lucky to have had a chance to work for and with include: Professor S. Liu, Professor J. Xie, Dr. J. Le Duff, Professor Dr. A. Richter, Dr. J. Bisognano, Dr. G. Neil, and Dr. F. Dylla. Because of them, my previous physics research career had been so enjoyable and fruitful. I’d like to mention that my current career as a software performance engineer has benefited tremendously from my previous career as a physicist. Although I left physics research and jumped to computers and software about a decade ago, the spirit of pursuing a subject in a rigorous, quantitative, and objective manner cultivated through my earlier physics research career has never left me. I have had this hopelessly immutable habitue of trying to deal with every software performance issue as quantitatively as possible as if it were a physics research subject. I have been totally soaked with that spirit, which gives me the power and energy for pursuing every challenging software performance and scalability problem quantitatively, and thus this book—Software Performance and Scalability: A Quantitative Approach.

With my career as a software performance professional, I’d like to especially thank Pat Crain, who introduced me to applying queuing theory to solving software performance challenges. Pat also encouraged me to write my first research paper on software performance, which was presented and awarded the best paper award in the category of software performance at the 2004 CMG Conference held in Las Vegas. I owe a debt of gratitude to Keith Gordon, who was the VP of the software company I worked for. Keith had enthusiastically read draft versions of my papers prior to publication and had always encouraged me to publish and share my software performance experience with the greater software community. I also feel excited to mention one of my fellow software performance engineers, Mary Shun, who encouraged me to write a book on software performance someday. Many thanks and this is it, Mary!
Special thanks are also due to Engel Martin, one of the greatest software performance group managers I have ever worked for. While taking on a tremendous amount of managerial responsibilities, Engel has demonstrated an extremely sharp and accurate sense of software performance and scalability issues at high levels. The atmosphere Engel created within the group he manages has always made me feel comfortable to express my opinions freely and to make my own judgments objectively on technical issues, as I used to as a scientist, for which I am truly grateful.

I would like to take this opportunity to thank the nonanonymous yet never-met authors of some hundreds of books I bought in mathematics, physics, computers, and software. The books they wrote fed my knowledge-hungry mind at various stages of my careers.

I sincerely thank those anonymous referees who offered very useful opinions on how to make this book more valuable to the readers and more suitable as a textbook for an upper division undergraduate or graduate level course for students majoring in computers and software. Their very helpful suggestions have been incorporated in the various parts of this book.

I also owe thanks to Dr. Larry Bernstein, who kindly recommended my book proposal to Wiley. The structure of this book has been deeply influenced by his seminal works published by Wiley in the series of *Quantitative Software Engineering*.

Paul Petralia at Wiley-Interscience mentored me as a first-time book writer through each step of the entire process. It would have not been such a smooth process without Paul’s guidance and high professionalism. Michael Christian helped me achieve the next milestone for the book—to get it into production at Wiley—with his hard work and high efficiency. My production manager, Shirley Thomas, production editor, Christine Punzo, compositor (Techset Composition Ltd.), and illustration manager, Dean Gonzalez, at Wiley were the hard-working, efficient engines behind completing the last stage of publishing this book. Needless to say, without such a highly efficient and professional team at Wiley, my software performance experience accumulated over a decade would have still been scattered in various publications and work notes. So many thanks to everyone involved at Wiley.

I’d like to thank my wife, Sarah Chen, who sacrificed so much to take care of our newborn son, William, most of the time, in order to let me sit down and focus on writing this book using my weekends, nightly hours, and even vacation times.

You as a reader are greatly appreciated as well. Your interest in this book has shown your strong motivation to further the success of your company and also your willingness to help contain global warming by developing high-performance and highly scalable software that burns less electric power.

HENRY H. LIU
PERFORMANCE VERSUS SCALABILITY

Before we start, I think I owe you an explanation about what the difference is between performance and scalability for a software system. In a word, performance and scalability are about the scalable performance for a software system.

You might find different explanations about performance versus scalability from other sources. In my opinion, performance and scalability for a software system differ from and correlate to each other as follows:

- **Performance** measures how fast and efficiently a software system can complete certain computing tasks, while **scalability** measures the trend of performance with increasing load. There are two major types of computing tasks that are measured using different performance metrics. For OLTP (online transaction processing) type of computing tasks consisting of interactive user activities, the metric of **response time** is used to measure how fast a system can respond to the requests of the interactive users, whereas for noninteractive batch jobs, the metric of **throughput** is used to measure the number of transactions a system can complete over a time period. Performance and scalability are inseparable from each other. It doesn’t make sense to talk about scalability if a software system doesn’t perform. However, a software system may perform but not scale.

- For a given environment that consists of properly sized hardware, properly configured operating system, and dependent middleware, if the performance of a software system **deteriorates rapidly** with increasing load (number of users or volume of transactions) prior to reaching the intended load level, then it is...
not scalable and will eventually underperform. In other words, we hope that the performance of a software system would sustain as a flat curve with increasing load prior to reaching the intended load level, which is the ideal scalability one can expect. This kind of scalability issue, which is classified as type I scalability issue, can be overcome with proper optimizations and tunings, as will be discussed in this book.

- If the performance of a software system becomes unacceptable when reaching a certain load level with a given environment, but it cannot be improved even with upgraded and/or additional hardware, then it is said that the software is not scalable. This kind of scalability issue, which is classified as type II scalability issue, cannot be overcome without going through some major architectural operations, which should be avoided from the beginning at any cost.

Unfortunately, there is no panacea for solving all software performance and scalability challenges. The best strategy is to start with the basics, being guided by queuing theory as well as by application programming interface (API) profiling when coping with software performance and scalability problems. This book teaches how one can make the most out of this strategy in a quantitative approach.

Let’s begin with the first part—the basics.
Part 1

The Basics

*I went behind the scenes to look at the mechanism.*
—Charles Babbage, 1791–1871, the father of computing

The factors that can critically impact the performance and scalability of a software system are abundant. The three factors that have the most impact on the performance and scalability of a software system are the raw capabilities of the underlying hardware platform, the maturity of the underlying software platform (mainly the operating system, various device interface drivers, the supporting virtual machine stack, the run-time environment, etc.), and its own design and implementation. If the software system is an application system built on some middleware systems such as various database servers, application servers, Web servers, and any other types of third-party components, then the performance and scalability of such middleware systems can directly affect the performance and scalability of the application system.

Understanding the performance and scalability of a software system qualitatively should begin with a solid understanding of all the performance bits built into the modern computer systems as well as all the performance and scalability implications associated with the various modern software platforms and architectures. Understanding the performance and scalability of a software system quantitatively calls for a test framework that can be depended upon to provide reliable information about the true performance and scalability of the software system in question. These ideas motivated me to select the following three chapters for this part:

- Chapter 1—Hardware Platform
- Chapter 2—Software Platform
- Chapter 3—Testing Software Performance and Scalability
The material presented in these three chapters is by no means the cliché you have heard again and again. I have filled in each chapter with real-world case studies so that you can actually feel the performance and scalability pitches associated with each case quantitatively.
What performance a software system exhibits often solely depends on the raw speed of the underlying hardware platform, which is largely determined by the central processing unit (CPU) horsepower of a computer. What scalability a software system exhibits depends on the scalability of the architecture of the underlying hardware platform as well. I have had many experiences with customers who reported that slow performance of the software system was simply caused by the use of undersized hardware. It’s fair to say that hardware platform is the number one most critical factor in determining the performance and scalability of a software system. We’ll see in this chapter the two supporting case studies associated with the Intel® hyperthreading technology and new Intel multicore processor architecture.

As is well known, the astonishing advances of computers can be characterized quantitatively by Moore’s law. Intel co-founder Gordon E. Moore stated in his 1965 seminal paper that the density of transistors on a computer chip is increasing exponentially, doubling approximately every two years. The trend has continued for more than half a century and is not expected to stop for another decade at least.

The quantitative approach pioneered by Moore has been very effective in quantifying the advances of computers. It has been extended into other areas of computer and software engineering as well, to help refine the methodologies of developing better software and computer architectures [Bernstein and Yuhas, 2005; Laird and
Brennan, 2006; Gabarro, 2006; Hennessy and Patterson, 2007]. This book is an attempt to introduce quantitativeness into dealing with the challenges of software performance and scalability facing the software industry today.

To see how modern computers have become so powerful, let’s begin with the Turing machine.

1.1 TURING MACHINE

Although Charles Babbage (1791–1871) is known as the father of computing, the most original idea of a computing machine was described by Alan Turing more than seven decades ago in 1936. Turing was a mathematician and is often considered the father of modern computer science.

As shown in Figure 1.1, a Turing machine consists of the following four basic elements:

- A tape, which is divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. This tape is assumed to be infinitely long on both ends. It can be read or written.
- A head that can read and write symbols on the tape.
- A table of instructions that tell the machine what to do next, based on the current state of the machine and the symbols it is reading on the tape.
- A state register that stores the states of the machine.

A Turing machine has two assumptions: one is the unlimited storage space and the other is completing a task regardless of the amount of time it takes. As a theoretical model, it exhibits the great power of abstraction to the highest degree. To some extent, modern computers are as close to Turing machines as modern men are close to cavemen. It’s so amazing that today’s computers still operate on the same principles.