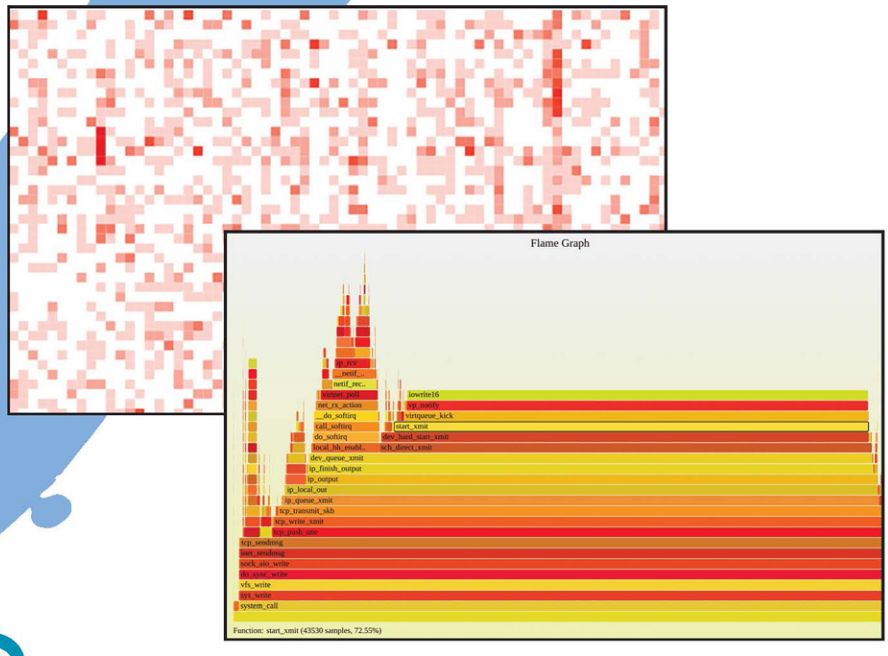
Systems Performance

Enterprise and the Cloud

Second Edition

Brendan Gregg



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Systems Performance

Second Edition

This page intentionally left blank

Systems Performance

Enterprise and the Cloud

Second Edition

Brendan Gregg

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2020944455

Cover images by Brendan Gregg
Page 9, Figure 1.5: Screenshot of System metrics GUI (Grafana) © 2020 Grafana Labs
Page 84, Figure 2.32: Screenshot of Firefox timeline chart © Netflix
Page 164, Figure 4.7: Screenshot of sar(1) sadf(1) SVG output © 2010 W3C
Page 560, Figure 10.12: Screenshot of Wireshark screenshot © Wireshark
Page 740, Figure 14.3: Screenshot of KernelShark © KernelShark

ISBN-13: 978-0-13-682015-4
ISBN-10: 0-13-682015-8

ScoutAutomatedPrintCode

Publisher
Mark L. Taub

Executive Editor
Greg Doench

Managing Producer
Sandra Schroeder

Sr. Content Producer
Julie B. Nahil

Project Manager
Rachel Paul

Copy Editor
Kim Wimpsett

Indexer
Ted Laux

Proofreader
Rachel Paul

Compositor
The CIP Group

For Deirdré Straughan,
an amazing person in technology,
and an amazing person—we did it.

This page intentionally left blank

Contents at a Glance

This page intentionally left blank

Contents

Preface

> *"There are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns—there are things we do not know we don't know."*
>
> —U.S. Secretary of Defense Donald Rumsfeld, February 12, 2002

While the previous statement was met with chuckles from those attending the press briefing, it summarizes an important principle that is as relevant in complex technical systems as it is in geopolitics: performance issues can originate from anywhere, including areas of the system that you know nothing about and you are therefore not checking (the unknown unknowns). This book may reveal many of these areas, while providing methodologies and tools for their analysis.

About This Edition

I wrote the first edition eight years ago and designed it to have a long shelf life. Chapters are structured to first cover durable skills (models, architecture, and methodologies) and then faster-changing skills (tools and tuning) as example implementations. While the example tools and tuning will go out of date, the durable skills show you how to stay updated.

There has been a large addition to Linux in the past eight years: Extended BPF, a kernel technology that powers a new generation of performance analysis tools, which is used by companies including Netflix and Facebook. I have included a BPF chapter and BPF tools in this new edition, and I have also published a deeper reference on the topic [Gregg 19]. The Linux perf and Ftrace tools have also seen many developments, and I have added separate chapters for them as well. The Linux kernel has gained many performance features and technologies, also covered. The hypervisors that drive cloud computing virtual machines, and container technologies, have also changed considerably; that content has been updated.

The first edition covered both Linux and Solaris equally. Solaris market share has shrunk considerably in the meantime [ITJobsWatch 20], so the Solaris content has been largely removed from this edition, making room for more Linux content to be included. However, your understanding of an operating system or kernel can be enhanced by considering an alternative, for perspective. For that reason, some mentions of Solaris and other operating systems are included in this edition.

For the past six years I have been a senior performance engineer at Netflix, applying the field of systems performance to the Netflix microservices environment. I've worked on the performance of hypervisors, containers, runtimes, kernels, databases, and applications. I've developed new methodologies and tools as needed, and worked with experts in cloud performance and Linux kernel engineering. These experiences have contributed to improving this edition.

About This Book

Welcome to *Systems Performance: Enterprise and the Cloud*, 2nd Edition! This book is about the performance of operating systems and of applications from the operating system context, and it is written for both enterprise server and cloud computing environments. Much of the material in this book can also aid your analysis of client devices and desktop operating systems. My aim is to help you get the most out of your systems, whatever they are.

When working with application software that is under constant development, you may be tempted to think of operating system performance—where the kernel has been developed and tuned for decades—as a solved problem. It isn't! The operating system is a complex body of software, managing a variety of ever-changing physical devices with new and different application workloads. The kernels are also in constant development, with features being added to improve the performance of particular workloads, and newly encountered bottlenecks being removed as systems continue to scale. Kernel changes such as the mitigations for the Meltdown vulnerability that were introduced in 2018 can also hurt performance. Analyzing and working to improve the performance of the operating system is an ongoing task that should lead to continual performance improvements. Application performance can also be analyzed from the operating system context to find more clues that might be missed using application-specific tools alone; I'll cover that here as well.

Operating System Coverage

The main focus of this book is the study of systems performance, using Linux-based operating systems on Intel processors as the primary example. The content is structured to help you study other kernels and processors as well.

Unless otherwise noted, the specific Linux distribution is not important in the examples used. The examples are mostly from the Ubuntu distribution and, when necessary, notes are included to explain differences for other distributions. The examples are also taken from a variety of system types: bare metal and virtualized, production and test, servers and client devices.

Across my career I've worked with a variety of different operating systems and kernels, and this has deepened my understanding of their design. To deepen your understanding as well, this book includes some mentions of Unix, BSD, Solaris, and Windows.

Other Content

Example screenshots from performance tools are included, not just for the data shown, but also to illustrate the types of data available. The tools often present the data in intuitive and self-explanatory ways, many in the familiar style of earlier Unix tools. This means that screenshots can be a powerful way to convey the purpose of these tools, some requiring little additional description. (If a tool does require laborious explanation, that may be a failure of design!)

Where it provides useful insight to deepen your understanding, I touch upon the history of certain technologies. It is also useful to learn a bit about the key people in this industry: you're likely to come across them or their work in performance and other contexts. A "who's who" list has been provided in Appendix E.

A handful of topics in this book were also covered in my prior book, *BPF Performance Tools* [Gregg 19]: in particular, BPF, BCC, bpftrace, tracepoints, kprobes, uprobes, and various BPF-based tools. You can refer to that book for more information. The summaries of these topics in this book are often based on that earlier book, and sometimes use the same text and examples.

What Isn't Covered

This book focuses on performance. To undertake all the example tasks given will require, at times, some system administration activities, including the installation or compilation of software (which is not covered here).

The content also summarizes operating system internals, which are covered in more detail in separate dedicated texts. Advanced performance analysis topics are summarized so that you are aware of their existence and can study them as needed from additional sources. See the Supplemental Material section at the end of this Preface.

How This Book Is Structured

Chapter 1, Introduction, is an introduction to systems performance analysis, summarizing key concepts and providing examples of performance activities.

Chapter 2, Methodologies, provides the background for performance analysis and tuning, including terminology, concepts, models, methodologies for observation and experimentation, capacity planning, analysis, and statistics.

Chapter 3, Operating Systems, summarizes kernel internals for the performance analyst. This is necessary background for interpreting and understanding what the operating system is doing.

Chapter 4, Observability Tools, introduces the types of system observability tools available, and the interfaces and frameworks upon which they are built.

Chapter 5, Applications, discusses application performance topics and observing them from the operating system.

Chapter 6, CPUs, covers processors, cores, hardware threads, CPU caches, CPU interconnects, device interconnects, and kernel scheduling.

Chapter 7, Memory, is about virtual memory, paging, swapping, memory architectures, buses, address spaces, and allocators.

Chapter 8, File Systems, is about file system I/O performance, including the different caches involved.

Chapter 9, Disks, covers storage devices, disk I/O workloads, storage controllers, RAID, and the kernel I/O subsystem.

Chapter 10, Network, is about network protocols, sockets, interfaces, and physical connections.

Chapter 11, Cloud Computing, introduces operating system– and hardware-based virtualization methods in common use for cloud computing, along with their performance overhead, isolation, and observability characteristics. This chapter covers hypervisors and containers.

Chapter 12, Benchmarking, shows how to benchmark accurately, and how to interpret others' benchmark results. This is a surprisingly tricky topic, and this chapter shows how you can avoid common mistakes and try to make sense of it.

Chapter 13, perf, summarizes the standard Linux profiler, perf(1), and its many capabilities. This is a reference to support perf(1)'s use throughout the book.

Chapter 14, Ftrace, summarizes the standard Linux tracer, Ftrace, which is especially suited for exploring kernel code execution.

Chapter 15, BPF, summarizes the standard BPF front ends: BCC and bpftrace.

Chapter 16, Case Study, contains a systems performance case study from Netflix, showing how a production performance puzzle was analyzed from beginning to end.

Chapters 1 to 4 provide essential background. After reading them, you can reference the remainder of the book as needed, in particular Chapters 5 to 12, which cover specific targets for analysis. Chapters 13 to 15 cover advanced profiling and tracing, and are optional reading for those who wish to learn one or more tracers in more detail.

Chapter 16 uses a storytelling approach to paint a bigger picture of a performance engineer's work. If you're new to performance analysis, you might want to read this first as an example of performance analysis using a variety of different tools, and then return to it when you've read the other chapters.

As a Future Reference

This book has been written to provide value for many years, by focusing on background and methodologies for the systems performance analyst.

To support this, many chapters have been separated into two parts. The first part consists of terms, concepts, and methodologies (often with those headings), which should stay relevant many years from now. The second provides examples of how the first part is implemented: architecture, analysis tools, and tunables, which, while they will become out-of-date, will still be useful as examples.

Tracing Examples

We frequently need to explore the operating system in depth, which can be done using tracing tools.

Since the first edition of this book, extended BPF has been developed and merged into the Linux kernel, powering a new generation of tracing tools that use the BCC and bpftrace front ends. This book focuses on BCC and bpftrace, and also the Linux kernel's built-in Ftrace tracer. BPF, BCC, and bpftrace, are covered in more depth in my prior book [Gregg 19].

Linux perf is also included in this book and is another tool that can do tracing. However, perf is usually included in chapters for its sampling and PMC analysis capabilities, rather than for tracing.

You may need or wish to use different tracing tools, which is fine. The tracing tools in this book are used to show the questions that you can ask of the system. It is often these questions, and the methodologies that pose them, that are the most difficult to know.

Intended Audience

The intended audience for this book is primarily systems administrators and operators of enterprise and cloud computing environments. It is also a reference for developers, database administrators, and web server administrators who need to understand operating system and application performance.

As a performance engineer at a company with a large compute environment (Netflix), I frequently work with SREs (site reliability engineers) and developers who are under enormous time pressure to solve multiple simultaneous performance issues. I have also been on the Netflix CORE SRE on-call rotation and have experienced this pressure firsthand. For many people, performance is not their primary job, and they need to know just enough to solve the current issues. Knowing that your time may be limited has encouraged me to keep this book as short as possible, and structure it to facilitate jumping ahead to specific chapters.

Another intended audience is students: this book is also suitable as a supporting text for a systems performance course. I have taught these classes before and learned which types of material work best in leading students to solve performance problems; that has guided my choice of content for this book.

Whether or not you are a student, the chapter exercises give you an opportunity to review and apply the material. These include some optional advanced exercises, which you are not expected to solve. (They may be impossible; they should at least be thought-provoking.)

In terms of company size, this book should contain enough detail to satisfy environments from small to large, including those with dozens of dedicated performance staff. For many smaller companies, the book may serve as a reference when needed, with only some portions of it used day to day.

Typographic Conventions

The following typographical conventions are used throughout this book:

Example	Description
netif_receive_skb()	Function name
iostat(1)	A command referenced by chapter 1 of its man page
read(2)	A system call referenced by its man page
malloc(3)	A C library function call referenced by its man page
vmstat(8)	An administration command referenced by its man page
Documentation/...	Linux documentation in the Linux kernel source tree
kernel/...	Linux kernel source code
fs/...	Linux kernel source code, file systems
CONFIG_...	Linux kernel configuration option (Kconfig)
`r_await`	Command line input and output

Example	Description
`mpstat 1`	Highlighting of a typed command or key detail
#	Superuser (root) shell prompt
$	User (non-root) shell prompt
^C	A command was interrupted (Ctrl-C)
[...]	Truncation

Supplemental Material, References, and Bibliography

References are listed are at the end of each chapter rather than in a single bibliography, allowing you to browse references related to each chapter's topic. The following selected texts can also be referenced for further background on operating systems and performance analysis:

[Jain 91] Jain, R., *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley, 1991.

[Vahalia 96] Vahalia, U., *UNIX Internals: The New Frontiers*, Prentice Hall, 1996.

[Cockcroft 98] Cockcroft, A., and Pettit, R., *Sun Performance and Tuning: Java and the Internet*, Prentice Hall, 1998.

[Musumeci 02] Musumeci, G. D., and Loukides, M., *System Performance Tuning*, 2nd Edition, O'Reilly, 2002.

[Bovet 05] Bovet, D., and Cesati, M., *Understanding the Linux Kernel,* 3rd Edition, O'Reilly, 2005.

[McDougall 06a] McDougall, R., Mauro, J., and Gregg, B., *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*, Prentice Hall, 2006.

[Gove 07] Gove, D., *Solaris Application Programming*, Prentice Hall, 2007.

[Love 10] Love, R., *Linux Kernel Development*, 3rd Edition, Addison-Wesley, 2010.

[Gregg 11a] Gregg, B., and Mauro, J., *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*, Prentice Hall, 2011.

[Gregg 13a] Gregg, B., *Systems Performance: Enterprise and the Cloud*, Prentice Hall, 2013 (first edition).

[Gregg 19] Gregg, B., *BPF Performance Tools: Linux System and Application Observability*, Addison-Wesley, 2019.

[ITJobsWatch 20] ITJobsWatch, "Solaris Jobs," https://www.itjobswatch.co.uk/jobs/uk/solaris.do#demand_trend, accessed 2020.

Acknowledgments

Thanks to all those who bought the first edition, especially those who made it recommended or required reading at their companies. Your support for the first edition has led to the creation of this second edition. Thank you.

This is the latest book on systems performance, but not the first. I'd like to thank prior authors for their work, work that I have built upon and referenced in this text. In particular I'd like to thank Adrian Cockcroft, Jim Mauro, Richard McDougall, Mike Loukides, and Raj Jain. As they have helped me, I hope to help you.

I'm grateful for everyone who provided feedback on this edition:

Deirdré Straughan has again supported me in various ways throughout this book, including using her years of experience in technical copy editing to improve every page. The words you read are from both of us. We enjoy not just spending time together (we are married now), but also working together. Thank you.

Philipp Marek is an IT forensics specialist, IT architect, and performance engineer at the Austrian Federal Computing Center. He provided early technical feedback on every topic in this book (an amazing feat) and even spotted problems in the first edition text. Philipp started programming in 1983 on a 6502, and has been looking for additional CPU cycles ever since. Thanks, Philipp, for your expertise and relentless work.

Dale Hamel (Shopify) also reviewed every chapter, providing important insights for various cloud technologies, and another consistent point of view across the entire book. Thanks for taking this on, Dale—right after helping with the BPF book!

Daniel Borkmann (Isovalent) provided deep technical feedback for a number of chapters, especially the networking chapter, helping me to better understand the complexities and technologies involved. Daniel is a Linux kernel maintainer with years of experience working on the kernel network stack and extended BPF. Thank you, Daniel, for the expertise and rigor.

I'm especially thankful that perf maintainer Arnaldo Carvalho de Melo (Red Hat) helped with Chapter 13, perf; and Ftrace creator Steven Rostedt (VMware) helped with Chapter 14, Ftrace, two topics that I had not covered well enough in the first edition. Apart from their help with this book, I also appreciate their excellent work on these advanced performance tools, tools that I've used to solve countless production issues at Netflix.

It has been a pleasure to have Dominic Kay pick through several chapters and find even more ways to improve their readability and technical accuracy. Dominic also helped with the first edition (and before that, was my colleague at Sun Microsystems working on performance). Thank you, Dominic.

My current performance colleague at Netflix, Amer Ather, provided excellent feedback on several chapters. Amer is a go-to engineer for understanding complex technologies. Zachary Jones (Verizon) also provided feedback for complex topics, and shared his performance expertise to improve the book. Thank you, Amer and Zachary.

A number of reviewers took on multiple chapters and engaged in discussion on specific topics: Alejandro Proaño (Amazon), Bikash Sharma (Facebook), Cory Lueninghoener (Los Alamos

National Laboratory), Greg Dunn (Amazon), John Arrasjid (Ottometric), Justin Garrison (Amazon), Michael Hausenblas (Amazon), and Patrick Cable (Threat Stack). Thanks, all, for your technical help and enthusiasm for the book.

Also thanks to Aditya Sarwade (Facebook), Andrew Gallatin (Netflix), Bas Smit, George Neville-Neil (JUUL Labs), Jens Axboe (Facebook), Joel Fernandes (Google), Randall Stewart (Netflix), Stephane Eranian (Google), and Toke Høiland-Jørgensen (Red Hat), for answering questions and timely technical feedback.

The contributors to my earlier book, *BPF Performance Tools*, have indirectly helped, as some material in this edition is based on that earlier book. That material was improved thanks to Alastair Robertson (Yellowbrick Data), Alexei Starovoitov (Facebook), Daniel Borkmann, Jason Koch (Netflix), Mary Marchini (Netflix), Masami Hiramatsu (Linaro), Mathieu Desnoyers (EfficiOS), Yonghong Song (Facebook), and more. See that book for the full acknowledgments.

This second edition builds upon the work in the first edition. The acknowledgments from the first edition thanked the many people who supported and contributed to that work; in summary, across multiple chapters I had technical feedback from Adam Leventhal, Carlos Cardenas, Darryl Gove, Dominic Kay, Jerry Jelinek, Jim Mauro, Max Bruning, Richard Lowe, and Robert Mustacchi. I also had feedback and support from Adrian Cockcroft, Bryan Cantrill, Dan McDonald, David Pacheco, Keith Wesolowski, Marsell Kukuljevic-Pearce, and Paul Eggleton. Roch Bourbonnais and Richard McDougall helped indirectly as I learned so much from their prior performance engineering work, and Jason Hoffman helped behind the scenes to make the first edition possible.

The Linux kernel is complicated and ever-changing, and I appreciate the stellar work by Jonathan Corbet and Jake Edge of lwn.net for summarizing so many deep topics. Many of their articles are referenced in this book.

A special thanks to Greg Doench, executive editor at Pearson, for his help, encouragement, and flexibility in making this process more efficient than ever. Thanks to content producer Julie Nahil (Pearson) and project manager Rachel Paul, for their attention to detail and help in delivering a quality book. Thanks to copy editor Kim Wimpsett for the working through another one of my lengthy and deeply technical books, finding many ways to improve the text.

And thanks, Mitchell, for your patience and understanding.

Since the first edition, I've continued to work as a performance engineer, debugging issues everywhere in the stack, from applications to metal. I now have many new experiences with performance tuning hypervisors, analyzing runtimes including the JVM, using tracers including Ftrace and BPF in production, and coping with the fast pace of changes in the Netflix microservices environment and the Linux kernel. So much of this is not well documented, and it had been daunting to consider what I needed to do for this edition. But I like a challenge.

About the Author

Brendan Gregg is an industry expert in computing performance and cloud computing. He is a senior performance architect at Netflix, where he does performance design, evaluation, analysis, and tuning. The author of multiple technical books, including *BPF Performance Tools*, he received the USENIX LISA Award for Outstanding Achievement in System Administration. He has also been a kernel engineer, performance lead, and professional technical trainer, and was program co-chair for the USENIX LISA 2018 conference. He has created performance tools included in multiple operating systems, along with visualizations and methodologies for performance analysis, including flame graphs.

This page intentionally left blank

Chapter 1

Introduction

Computer performance is an exciting, varied, and challenging discipline. This chapter introduces you to the field of systems performance. The learning objectives of this chapter are:

- Understand systems performance, roles, activities, and challenges.
- Understand the difference between observability and experimental tools.
- Develop a basic understanding of performance observability: statistics, profiling, flame graphs, tracing, static instrumentation, and dynamic instrumentation.
- Learn the role of methodologies and the Linux 60-second checklist.

References to later chapters are included so that this works as an introduction both to systems performance and to this book. This chapter finishes with case studies to show how systems performance works in practice.

1.1 Systems Performance

Systems performance studies the performance of an entire computer system, including all major software and hardware components. Anything in the data path, from storage devices to application software, is included, because it can affect performance. For distributed systems this means multiple servers and applications. If you don't have a diagram of your environment showing the data path, find one or draw it yourself; this will help you understand the relationships between components and ensure that you don't overlook entire areas.

The typical goals of systems performance are to improve the end-user experience by reducing latency and to reduce computing cost. Reducing cost can be achieved by eliminating inefficiencies, improving system throughput, and general tuning.

Figure 1.1 shows a generic system software stack on a single server, including the operating system (OS) kernel, with example database and application tiers. The term *full stack* is sometimes used to describe only the application environment, including databases, applications, and web servers. When speaking of systems performance, however, we use *full stack* to mean the entire software stack from the application down to metal (the hardware), including system libraries, the kernel, and the hardware itself. Systems performance studies the full stack.

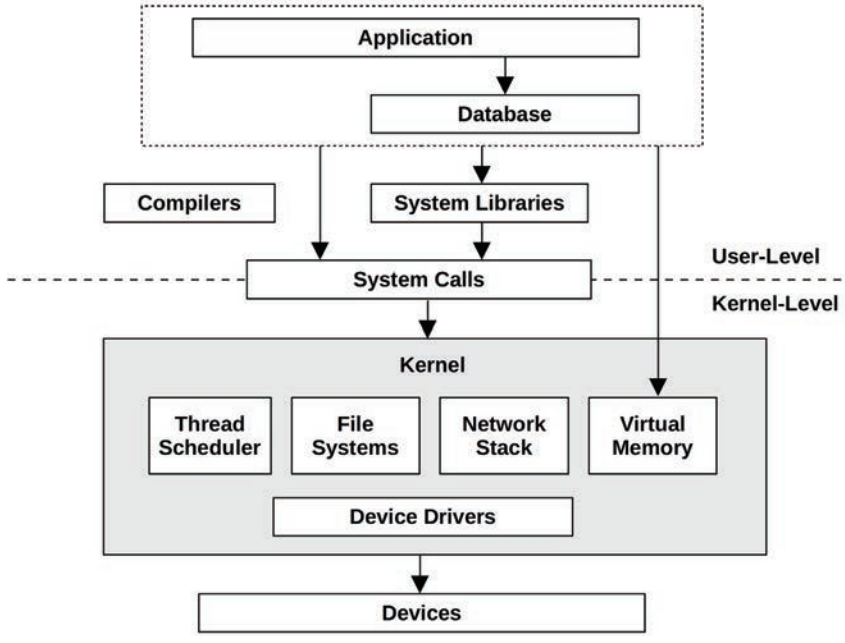


Figure 1.1 Generic system software stack

Compilers are included in Figure 1.1 because they play a role in systems performance. This stack is discussed in Chapter 3, Operating Systems, and investigated in more detail in later chapters. The following sections describe systems performance in more detail.

1.2 Roles

Systems performance is done by a variety of job roles, including system administrators, site reliability engineers, application developers, network engineers, database administrators, web administrators, and other support staff. For many of these roles, performance is only part of the job, and performance analysis focuses on that role's area of responsibility: the network team checks the network, the database team checks the database, and so on. For some performance issues, finding the root cause or contributing factors requires a cooperative effort from more than one team.

Some companies employ *performance engineers*, whose primary activity is performance. They can work with multiple teams to perform a holistic study of the environment, an approach that may be vital in resolving complex performance issues. They can also act as a central resource to find and develop better tooling for performance analysis and capacity planning across the whole environment.

For example, Netflix has a cloud performance team, of which I am a member. We assist the microservice and SRE teams with performance analysis and build performance tools for everyone to use.

Companies that hire multiple performance engineers can allow individuals to specialize in one or more areas, providing deeper levels of support. For example, a large performance engineering team may include specialists in kernel performance, client performance, language performance (e.g., Java), runtime performance (e.g., the JVM), performance tooling, and more.

1.3 Activities

Systems performance involves a variety of activities. The following is a list of activities that are also ideal steps in the life cycle of a software project from conception through development to production deployment. Methodologies and tools to help perform these activities are covered in this book.

1. Setting performance objectives and performance modeling for a future product.
2. Performance characterization of prototype software and hardware.
3. Performance analysis of in-development products in a test environment.
4. Non-regression testing for new product versions.
5. Benchmarking product releases.
6. Proof-of-concept testing in the target production environment.
7. Performance tuning in production.
8. Monitoring of running production software.
9. Performance analysis of production issues.
10. Incident reviews for production issues.
11. Performance tool development to enhance production analysis.

Steps 1 to 5 comprise traditional product development, whether for a product sold to customers or a company-internal service. The product is then launched, perhaps first with proof-of-concept testing in the target environment (customer or local), or it may go straight to deployment and configuration. If an issue is encountered in the target environment (steps 6 to 9), it means that the issue was not detected or fixed during the development stages.

Performance engineering should ideally begin before any hardware is chosen or software is written: the first step should be to set objectives and create a performance model. However, products are often developed without this step, deferring performance engineering work to a later time, after a problem arises. With each step of the development process it can become progressively harder to fix performance issues that arise due to architectural decisions made earlier.

Cloud computing provides new techniques for proof-of-concept testing (step 6) that encourage skipping the earlier steps (steps 1 to 5). One such technique is testing new software on a single instance with a fraction of the production workload: this is known as *canary testing*. Another technique makes this a normal step in software deployment: traffic is gradually moved to a new pool of instances while leaving the old pool online as a backup; this is known as *blue-green*

deployment.[1] With such safe-to-fail options available, new software is often tested in production without any prior performance analysis, and quickly reverted if need be. I recommend that, when practical, you also perform the earlier activities so that the best performance can be achieved (although there may be time-to-market reasons for moving to production sooner).

The term *capacity planning* can refer to a number of the preceding activities. During design, it includes studying the resource footprint of development software to see how well the design can meet the target needs. After deployment, it includes monitoring resource usage to predict problems before they occur.

The performance analysis of production issues (step 9) may also involve site reliability engineers (SREs); this step is followed by incident review meetings (step 10) to analyze what happened, share debugging techniques, and look for ways to avoid the same incident in the future. Such meetings are similar to developer *retrospectives* (see [Corry 20] for retrospectives and their anti-patterns).

Environments and activities vary from company to company and product to product, and in many cases not all ten steps are performed. Your job may also focus on only some or just one of these activities.

1.4 Perspectives

Apart from a focus on different activities, performance roles can be viewed from different perspectives. Two perspectives for performance analysis are labeled in Figure 1.2: *workload analysis* and *resource analysis*, which approach the software stack from different directions.

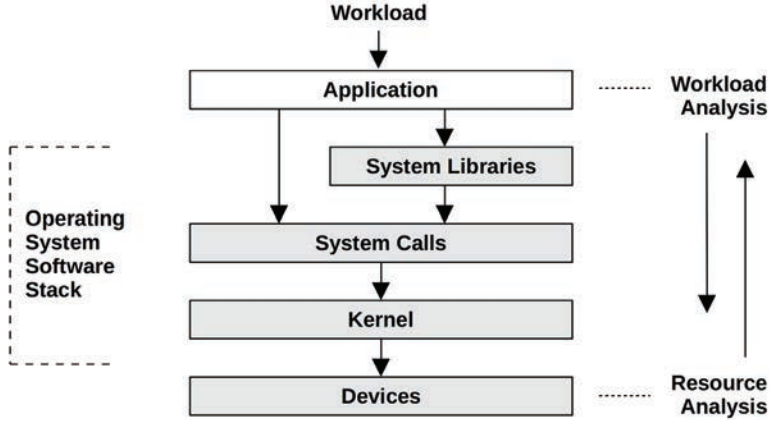


Figure 1.2 Analysis perspectives

The resource analysis perspective is commonly employed by system administrators, who are responsible for the system resources. Application developers, who are responsible for the

[1] Netflix uses the terminology *red-black deployments*.

delivered performance of the workload, commonly focus on the workload analysis perspective. Each perspective has its own strengths, discussed in detail in Chapter 2, Methodologies. For challenging issues, it helps to try analyzing from both perspectives.

1.5 Performance Is Challenging

Systems performance engineering is a challenging field for a number of reasons, including that it is subjective, it is complex, there may not be a single root cause, and it often involves multiple issues.

1.5.1 Subjectivity

Technology disciplines tend to be *objective*, so much so that people in the industry are known for seeing things in black and white. This can be true of software troubleshooting, where a bug is either present or absent and is either fixed or not fixed. Such bugs often manifest as error messages that can be easily interpreted and understood to mean the presence of an error.

Performance, on the other hand, is often *subjective*. With performance issues, it can be unclear whether there is an issue to begin with, and if so, when it has been fixed. What may be considered "bad" performance for one user, and therefore an issue, may be considered "good" performance for another.

Consider the following information:

> The average disk I/O response time is 1 ms.

Is this "good" or "bad"? While response time, or latency, is one of the best metrics available, interpreting latency information is difficult. To some degree, whether a given metric is "good" or "bad" may depend on the performance expectations of the application developers and end users.

Subjective performance can be made objective by defining clear goals, such as having a target average response time, or requiring a percentage of requests to fall within a certain latency range. Other ways to deal with this subjectivity are introduced in Chapter 2, Methodologies, including latency analysis.

1.5.2 Complexity

In addition to subjectivity, performance can be a challenging discipline due to the complexity of systems and the lack of an obvious starting point for analysis. In cloud computing environments you may not even know which server instance to look at first. Sometimes we begin with a hypothesis, such as blaming the network or a database, and the performance analyst must figure out if this is the right direction.

Performance issues may also originate from complex interactions between subsystems that perform well when analyzed in isolation. This can occur due to a *cascading failure*, when one failed component causes performance issues in others. To understand the resulting issue, you must untangle the relationships between components and understand how they contribute.

Bottlenecks can also be complex and related in unexpected ways; fixing one may simply move the bottleneck elsewhere in the system, with overall performance not improving as much as hoped.

Apart from the complexity of the system, performance issues may also be caused by a complex characteristic of the production workload. These cases may never be reproducible in a lab environment, or only intermittently so.

Solving complex performance issues often requires a holistic approach. The whole system—both its internals and its external interactions—may need to be investigated. This requires a wide range of skills, and can make performance engineering a varied and intellectually challenging line of work.

Different methodologies can be used to guide us through these complexities, as introduced in Chapter 2; Chapters 6 to 10 include specific methodologies for specific system resources: CPUs, Memory, File Systems, Disks, and Network. (The analysis of complex systems in general, including oil spills and the collapse of financial systems, has been studied by [Dekker 18].)

In some cases, a performance issue can be caused by the interaction of these resources.

1.5.3 Multiple Causes

Some performance issues do not have a single root cause, but instead have multiple contributing factors. Imagine a scenario where three normal events occur simultaneously and combine to cause a performance issue: each is a normal event that in isolation is not the root cause.

Apart from multiple causes, there can also be multiple performance issues.

1.5.4 Multiple Performance Issues

Finding *a* performance issue is usually not the problem; in complex software there are often many. To illustrate this, try finding the bug database for your operating system or applications and search for the word *performance*. You might be surprised! Typically, there will be a number of performance issues that are known but not yet fixed, even in mature software that is considered to have high performance. This poses yet another difficulty when analyzing performance: the real task isn't finding an issue; it's identifying which issue or issues matter *the most*.

To do this, the performance analyst must *quantify* the magnitude of issues. Some performance issues may not apply to your workload, or may apply only to a very small degree. Ideally, you will not just quantify the issues but also estimate the potential speedup to be gained for each one. This information can be valuable when management looks for justification for spending engineering or operations resources.

A metric well suited to performance quantification, when available, is *latency*.

1.6 Latency

Latency is a measure of time spent waiting, and is an essential performance metric. Used broadly, it can mean the time for any operation to complete, such as an application request, a database query, a file system operation, and so forth. For example, latency can express the time

for a website to load completely, from link click to screen paint. This is an important metric for both the customer and the website provider: high latency can cause frustration, and customers may take their business elsewhere.

As a metric, latency can allow maximum speedup to be estimated. For example, Figure 1.3 depicts a database query that takes 100 ms (which is the latency) during which it spends 80 ms blocked waiting for disk reads. The maximum performance improvement by eliminating disk reads (e.g., by caching) can be calculated: from 100 ms to 20 ms (100 – 80) is five times (5x) faster. This is the estimated *speedup*, and the calculation has also quantified the performance issue: disk reads are causing the query to run up to 5x more slowly.

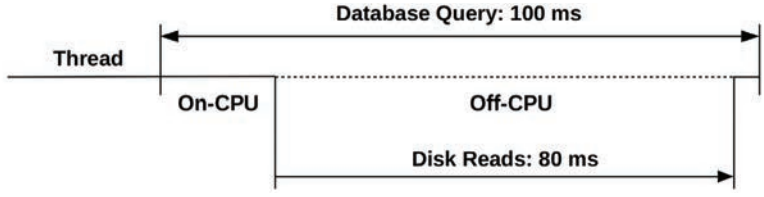


Figure 1.3 Disk I/O latency example

Such a calculation is not possible when using other metrics. I/O operations per second (IOPS), for example, depend on the type of I/O and are often not directly comparable. If a change were to reduce the IOPS rate by 80%, it is difficult to know what the performance impact would be. There might be 5x fewer IOPS, but what if each of these I/O increased in size (bytes) by 10x?

Latency can also be ambiguous without qualifying terms. For example, in networking, latency can mean the time for a connection to be established but not the data transfer time; or it can mean the total duration of a connection, including the data transfer (e.g., DNS latency is commonly measured this way). Throughout this book I will use clarifying terms where possible: those examples would be better described as *connection* latency and *request* latency. Latency terminology is also summarized at the beginning of each chapter.

While latency is a useful metric, it hasn't always been available when and where needed. Some system areas provide average latency only; some provide no latency measurements at all. With the availability of new BPF[2]-based observability tools, latency can now be measured from custom arbitrary points of interest and can provide data showing the full distribution of latency.

1.7 Observability

Observability refers to understanding a system through observation, and classifies the tools that accomplish this. This includes tools that use counters, profiling, and tracing. It does not include benchmark tools, which modify the state of the system by performing a workload *experiment*. For production environments, observability tools should be tried first wherever possible, as experimental tools may perturb production workloads through resource contention. For test environments that are idle, you may wish to begin with benchmarking tools to determine hardware performance.

[2] BPF is now a name and no longer an acronym (originally Berkeley Packet Filter).

In this section I'll introduce counters, metrics, profiling, and tracing. I'll explain observability in more detail in Chapter 4, covering system-wide versus per-process observability, Linux observability tools, and their internals. Chapters 5 to 11 include chapter-specific sections on observability, for example, Section 6.6 for CPU observability tools.

1.7.1 Counters, Statistics, and Metrics

Applications and the kernel typically provide data on their state and activity: operation counts, byte counts, latency measurements, resource utilization, and error rates. They are typically implemented as integer variables called *counters* that are hard-coded in the software, some of which are cumulative and always increment. These cumulative counters can be read at different times by performance tools for calculating *statistics*: the rate of change over time, the average, percentiles, etc.

For example, the vmstat(8) utility prints a system-wide summary of virtual memory statistics and more, based on kernel counters in the /proc file system. This example vmstat(8) output is from a 48-CPU production API server:

```
$ vmstat 1 5
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
19  0      0 6531592  42656 1672040    0    0     1     7   21   33 51  4 46  0  0
26  0      0 6533412  42656 1672064    0    0     0     0 81262 188942 54  4 43  0  0
62  0      0 6533856  42656 1672088    0    0     0     8 80865 180514 53  4 43  0  0
34  0      0 6532972  42656 1672088    0    0     0     0 81250 180651 53  4 43  0  0
31  0      0 6534876  42656 1672088    0    0     0     0 74389 168210 46  3 51  0  0
```

This shows a system-wide CPU utilization of around 57% (cpu us + sy columns). The columns are explained in detail in Chapters 6 and 7.

A *metric* is a statistic that has been selected to evaluate or monitor a target. Most companies use monitoring agents to record selected statistics (metrics) at regular intervals, and chart them in a graphical interface to see changes over time. Monitoring software can also support creating custom *alerts* from these metrics, such as sending emails to notify staff when problems are detected.

This hierarchy from counters to alerts is depicted in Figure 1.4. Figure 1.4 is provided as a guide to help you understand these terms, but their use in the industry is not rigid. The terms *counters*, *statistics*, and *metrics* are often used interchangeably. Also, alerts may be generated by any layer, and not just a dedicated alerting system.

As an example of graphing metrics, Figure 1.5 is a screenshot of a Grafana-based tool observing the same server as the earlier vmstat(8) output.

These line graphs are useful for capacity planning, helping you predict when resources will become exhausted.

Your interpretation of performance statistics will improve with an understanding of how they are calculated. Statistics, including averages, distributions, modes, and outliers, are summarized in Chapter 2, Methodologies, Section 2.8, Statistics.

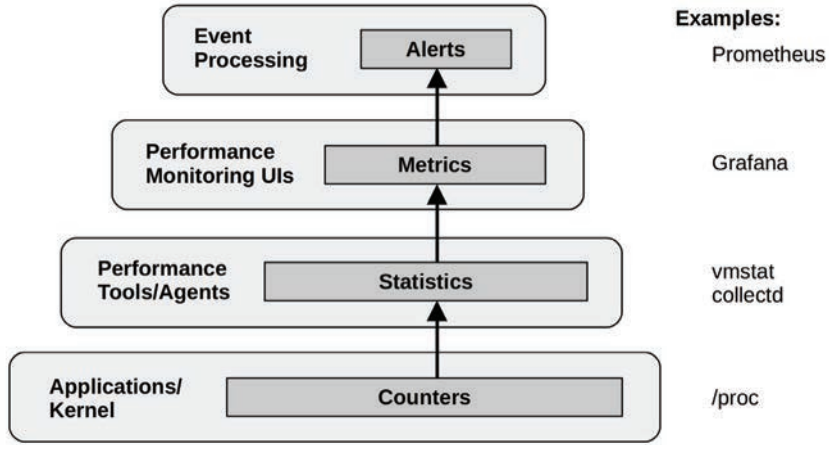


Figure 1.4 Performance instrumentation terminology

Figure 1.5 System metrics GUI (Grafana)

Sometimes, time-series metrics are all that is needed to resolve a performance issue. Knowing the exact time a problem began may correlate with a known software or configuration change, which can be reverted. Other times, metrics only point in a direction, suggesting that there is a CPU or disk issue, but without explaining why. Profiling or tracing tools are necessary to dig deeper and find the cause.

1.7.2 Profiling

In systems performance, the term *profiling* usually refers to the use of tools that perform sampling: taking a subset (a sample) of measurements to paint a coarse picture of the target. CPUs are a common profiling target. The commonly used method to profile CPUs involves taking timed-interval samples of the on-CPU code paths.

An effective visualization of CPU profiles is *flame graphs*. CPU flame graphs can help you find more performance wins than any other tool, after metrics. They reveal not only CPU issues, but other types of issues as well, found by the CPU footprints they leave behind. Issues of lock contention can be found by looking for CPU time in spin paths; memory issues can be analyzed by finding excessive CPU time in memory allocation functions (malloc()), along with the code paths that led to them; performance issues involving misconfigured networking may be discovered by seeing CPU time in slow or legacy codepaths; and so on.

Figure 1.6 is an example CPU flame graph showing the CPU cycles spent by the iperf(1) network micro-benchmark tool.

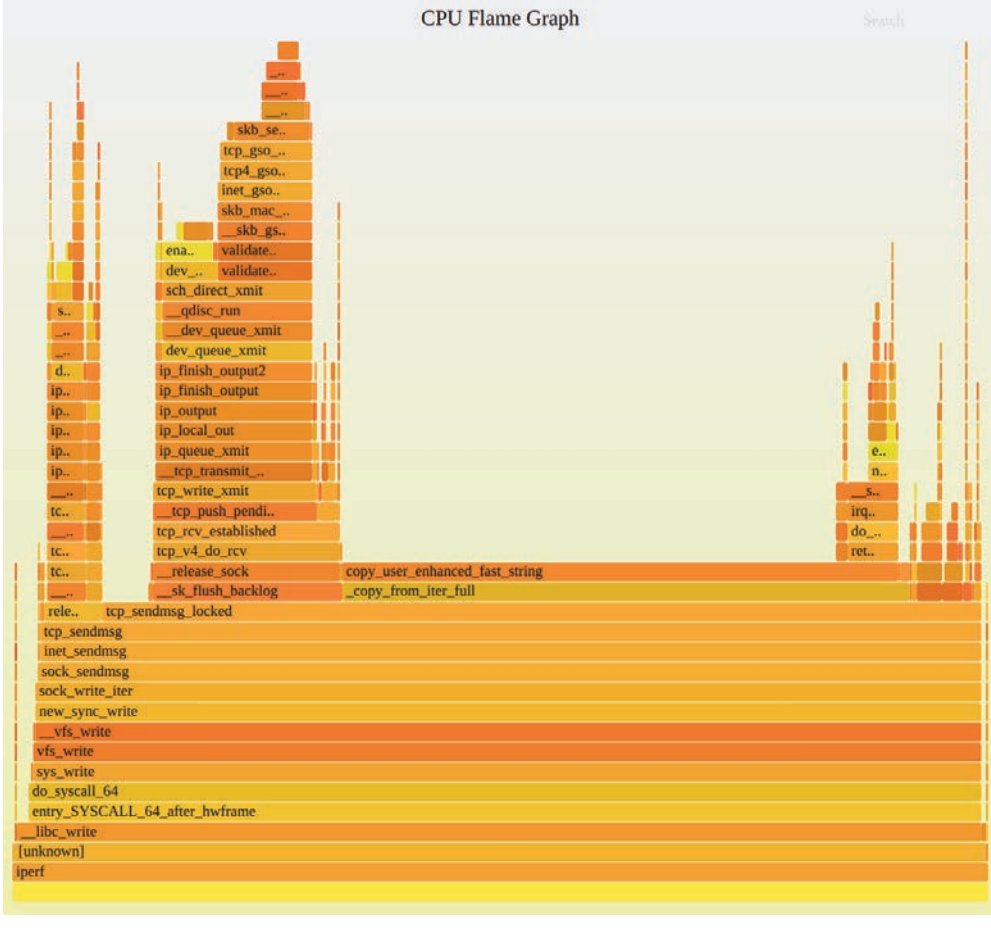


Figure 1.6 CPU profiling using flame graphs

This flame graph shows how much CPU time is spent copying bytes (the path that ends in copy_user_enhanced_fast_string()) versus TCP transmission (the tower on the left that includes tcp_write_xmit()). The widths are proportional to the CPU time spent, and the vertical axis shows the code path.

Profilers are explained in Chapters 4, 5, and 6, and the flame graph visualization is explained in Chapter 6, CPUs, Section 6.7.3, Flame Graphs.

1.7.3 Tracing

Tracing is event-based recording, where event data is captured and saved for later analysis or consumed on-the-fly for custom summaries and other actions. There are special-purpose tracing tools for system calls (e.g., Linux strace(1)) and network packets (e.g., Linux tcpdump(8)); and general-purpose tracing tools that can analyze the execution of all software and hardware events (e.g., Linux Ftrace, BCC, and bpftrace). These all-seeing tracers use a variety of event sources, in particular, static and dynamic instrumentation, and BPF for programmability.

Static Instrumentation

Static instrumentation describes hard-coded software instrumentation points added to the source code. There are hundreds of these points in the Linux kernel that instrument disk I/O, scheduler events, system calls, and more. The Linux technology for kernel static instrumentation is called *tracepoints*. There is also a static instrumentation technology for user-space software called *user statically defined tracing* (USDT). USDT is used by libraries (e.g., libc) for instrumenting library calls and by many applications for instrumenting service requests.

As an example tool that uses static instrumentation, execsnoop(8) prints new processes created while it is tracing (running) by instrumenting a tracepoint for the execve(2) system call. The following shows execsnoop(8) tracing an SSH login:

```
# execsnoop
PCOMM            PID    PPID   RET ARGS
ssh              30656  20063    0 /usr/bin/ssh 0
sshd             30657  1401     0 /usr/sbin/sshd -D -R
sh               30660  30657    0
env              30661  30660    0 /usr/bin/env -i PATH=/usr/local/sbin:/usr/local...
run-parts        30661  30660    0 /bin/run-parts --lsbsysinit /etc/update-motd.d
00-header        30662  30661    0 /etc/update-motd.d/00-header
uname            30663  30662    0 /bin/uname -o
uname            30664  30662    0 /bin/uname -r
uname            30665  30662    0 /bin/uname -m
10-help-text     30666  30661    0 /etc/update-motd.d/10-help-text
50-motd-news     30667  30661    0 /etc/update-motd.d/50-motd-news
cat              30668  30667    0 /bin/cat /var/cache/motd-news
cut              30671  30667    0 /usr/bin/cut -c -80
tr               30670  30667    0 /usr/bin/tr -d \000-\011\013\014\016-\037
head             30669  30667    0 /usr/bin/head -n 10
```

```
80-esm           30672  30661    0 /etc/update-motd.d/80-esm
lsb_release      30673  30672    0 /usr/bin/lsb_release -cs
[...]
```

This is especially useful for revealing short-lived processes that may be missed by other observability tools such as top(1). These short-lived processes can be a source of performance issues.

See Chapter 4 for more information about tracepoints and USDT probes.

Dynamic Instrumentation

Dynamic instrumentation creates instrumentation points after the software is running, by modifying in-memory instructions to insert instrumentation routines. This is similar to how debuggers can insert a breakpoint on any function in running software. Debuggers pass execution flow to an interactive debugger when the breakpoint is hit, whereas dynamic instrumentation runs a routine and then continues the target software. This capability allows custom performance statistics to be created from any running software. Issues that were previously impossible or prohibitively difficult to solve due to a lack of observability can now be fixed.

Dynamic instrumentation is so different from traditional observation that it can be difficult, at first, to grasp its role. Consider an operating system kernel: analyzing kernel internals can be like venturing into a dark room, with candles (system counters) placed where the kernel engineers thought they were needed. Dynamic instrumentation is like having a flashlight that you can point anywhere.

Dynamic instrumentation was first created in the 1990s [Hollingsworth 94], along with tools that use it called *dynamic tracers* (e.g., kerninst [Tamches 99]). For Linux, dynamic instrumentation was first developed in 2000 [Kleen 08] and began merging into the kernel in 2004 (kprobes). However, these technologies were not well known and were difficult to use. This changed when Sun Microsystems launched their own version in 2005, DTrace, which was easy to use and production-safe. I developed many DTrace-based tools that showed how important it was for systems performance, tools that saw widespread use and helped make DTrace and dynamic instrumentation well-known.

BPF

BPF, which originally stood for Berkeley Packet Filter, is powering the latest dynamic tracing tools for Linux. BPF originated as a mini in-kernel virtual machine for speeding up the execution of tcpdump(8) expressions. Since 2013 it has been extended (hence is sometimes called eBPF[3]) to become a generic in-kernel execution environment, one that provides safety and fast access to resources. Among its many new uses are tracing tools, where it provides programmability for the BPF Compiler Collection (BCC) and bpftrace front ends. execsnoop(8), shown earlier, is a BCC tool.[4]

[3] eBPF was initially used to describe this extended BPF; however, the technology is now referred to as just BPF.

[4] I first developed it for DTrace, and I have since developed it for other tracers including BCC and bpftrace.

Chapter 3 explains BPF, and Chapter 15 introduces the BPF tracing front ends: BCC and bpftrace. Other chapters introduce many BPF-based tracing tools in their observability sections; for example, CPU tracing tools are included in Chapter 6, CPUs, Section 6.6, Observability Tools. I have also published prior books on tracing tools (for DTrace [Gregg 11a] and BPF [Gregg 19]).

Both perf(1) and Ftrace are also tracers with some similar capabilities to the BPF front ends. perf(1) and Ftrace are covered in Chapters 13 and 14.

1.8 Experimentation

Apart from observability tools there are also *experimentation* tools, most of which are benchmarking tools. These perform an experiment by applying a synthetic workload to the system and measuring its performance. This must be done carefully, because experimental tools can perturb the performance of systems under test.

There are *macro-benchmark* tools that simulate a real-world workload such as clients making application requests; and there are *micro-benchmark* tools that test a specific component, such as CPUs, disks, or networks. As an analogy: a car's lap time at Laguna Seca Raceway could be considered a macro-benchmark, whereas its top speed and 0 to 60mph time could be considered micro-benchmarks. Both benchmark types are important, although micro-benchmarks are typically easier to debug, repeat, and understand, and are more stable.

The following example uses iperf(1) on an idle server to perform a TCP network throughput micro-benchmark with a remote idle server. This benchmark ran for ten seconds (`-t 10`) and produces per-second averages (`-i 1`):

```
# iperf -c 100.65.33.90 -i 1 -t 10
------------------------------------------------------------
Client connecting to 100.65.33.90, TCP port 5001
TCP window size: 12.0 MByte (default)
------------------------------------------------------------
[  3] local 100.65.170.28 port 39570 connected with 100.65.33.90 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec   582 MBytes  4.88 Gbits/sec
[  3]  1.0- 2.0 sec   568 MBytes  4.77 Gbits/sec
[  3]  2.0- 3.0 sec   574 MBytes  4.82 Gbits/sec
[  3]  3.0- 4.0 sec   571 MBytes  4.79 Gbits/sec
[  3]  4.0- 5.0 sec   571 MBytes  4.79 Gbits/sec
[  3]  5.0- 6.0 sec   432 MBytes  3.63 Gbits/sec
[  3]  6.0- 7.0 sec   383 MBytes  3.21 Gbits/sec
[  3]  7.0- 8.0 sec   388 MBytes  3.26 Gbits/sec
[  3]  8.0- 9.0 sec   390 MBytes  3.28 Gbits/sec
[  3]  9.0-10.0 sec   383 MBytes  3.22 Gbits/sec
[  3]  0.0-10.0 sec  4.73 GBytes  4.06 Gbits/sec
```

The output shows a throughput[5] of around 4.8 Gbits for the first five seconds, which drops to around 3.2 Gbits/sec. This is an interesting result that shows bi-modal throughput. To improve performance, one might focus on the 3.2 Gbits/sec mode, and search for other metrics that can explain it.

Consider the drawbacks of debugging this performance issue on a production server using observability tools alone. Network throughput can vary from second to second because of natural variance in the client workload, and the underlying bi-modal behavior of the network might not be apparent. By using iperf(1) with a fixed workload, you eliminate client variance, revealing the variance due to other factors (e.g., external network throttling, buffer utilization, and so on).

As I recommended earlier, on production systems you should first try observability tools. However, there are so many observability tools that you might spend hours working through them when an experimental tool would lead to quicker results. An analogy taught to me by a senior performance engineer (Roch Bourbonnais) many years ago was this: you have two hands, observability and experimentation. Only using one type of tool is like trying to solve a problem one-handed.

Chapters 6 to 10 include sections on experimental tools; for example, CPU experimental tools are covered in Chapter 6, CPUs, Section 6.8, Experimentation.

1.9 Cloud Computing

Cloud computing, a way to deploy computing resources on demand, has enabled rapid scaling of applications by supporting their deployment across an increasing number of small virtual systems called *instances*. This has decreased the need for rigorous capacity planning, as more capacity can be added from the cloud at short notice. In some cases it has also increased the desire for performance analysis, because using fewer resources can mean fewer systems. Since cloud usage is typically charged by the minute or hour, a performance win resulting in fewer systems can mean immediate cost savings. Compare this scenario to an enterprise data center, where you may be locked into a fixed support contract for years, unable to realize cost savings until the contract has ended.

New difficulties caused by cloud computing and virtualization include the management of performance effects from other tenants (sometimes called *performance isolation*) and physical system observability from each tenant. For example, unless managed properly by the system, disk I/O performance may be poor due to contention with a neighbor. In some environments, the true usage of the physical disks may not be observable by each tenant, making identification of this issue difficult.

These topics are covered in Chapter 11, Cloud Computing.

[5] The output uses the term "Bandwidth," a common misuse. Bandwidth refers to the maximum possible throughput, which iperf(1) is not measuring. iperf(1) is measuring the current rate of its network workload: its *throughput*.

1.10 Methodologies

Methodologies are a way to document the recommended steps for performing various tasks in systems performance. Without a methodology, a performance investigation can turn into a fishing expedition: trying random things in the hope of catching a win. This can be time-consuming and ineffective, while allowing important areas to be overlooked. Chapter 2, Methodologies, includes a library of methodologies for systems performance. The following is the first I use for any performance issue: a tool-based checklist.

1.10.1 Linux Perf Analysis in 60 Seconds

This is a Linux tool-based checklist that can be executed in the first 60 seconds of a performance issue investigation, using traditional tools that should be available for most Linux distributions [Gregg 15a]. Table 1.1 shows the commands, what to check for, and the section in this book that covers the command in more detail.

Table 1.1 **Linux 60-second analysis checklist**

#	Tool	Check	Section
1	`uptime`	Load averages to identify if load is increasing or decreasing (compare 1-, 5-, and 15-minute averages).	6.6.1
2	`dmesg -T | tail`	Kernel errors including OOM events.	7.5.11
3	`vmstat -SM 1`	System-wide statistics: run queue length, swapping, overall CPU usage.	7.5.1
4	`mpstat -P ALL 1`	Per-CPU balance: a single busy CPU can indicate poor thread scaling.	6.6.3
5	`pidstat 1`	Per-process CPU usage: identify unexpected CPU consumers, and user/system CPU time for each process.	6.6.7
6	`iostat -sxz 1`	Disk I/O statistics: IOPS and throughput, average wait time, percent busy.	9.6.1
7	`free -m`	Memory usage including the file system cache.	8.6.2
8	`sar -n DEV 1`	Network device I/O: packets and throughput.	10.6.6
9	`sar -n TCP,ETCP 1`	TCP statistics: connection rates, retransmits.	10.6.6
10	`top`	Check overview.	6.6.6

This checklist can also be followed using a monitoring GUI, provided the same metrics are available.[6]

[6] You could even make a custom dashboard for this checklist; however, bear in mind that this checklist was designed to make the most of readily available CLI tools, and monitoring products may have more (and better) metrics available. I'd be more inclined to make custom dashboards for the USE method and other methodologies.

Chapter 2, Methodologies, as well as later chapters, contain many more methodologies for performance analysis, including the USE method, workload characterization, latency analysis, and more.

1.11 Case Studies

If you are new to systems performance, case studies showing when and why various activities are performed can help you relate them to your current environment. Two hypothetical examples are summarized here; one is a performance issue involving disk I/O, and one is performance testing of a software change.

These case studies describe activities that are explained in other chapters of this book. The approaches described here are also intended to show not the right way or the only way, but rather *a* way that these performance activities can be conducted, for your critical consideration.

1.11.1 Slow Disks

Sumit is a system administrator at a medium-size company. The database team has filed a support ticket complaining of "slow disks" on one of their database servers.

Sumit's first task is to learn more about the issue, gathering details to form a problem statement. The ticket claims that the disks are slow, but it doesn't explain whether this is causing a database issue or not. Sumit responds by asking these questions:

- Is there currently a database performance issue? How is it measured?
- How long has this issue been present?
- Has anything changed with the database recently?
- Why were the disks suspected?

The database team replies: "We have a log for queries slower than 1,000 milliseconds. These usually don't happen, but during the past week they have been growing to dozens per hour. AcmeMon showed that the disks were busy."

This confirms that there is a real database issue, but it also shows that the disk hypothesis is likely a guess. Sumit wants to check the disks, but he also wants to check other resources quickly in case that guess was wrong.

AcmeMon is the company's basic server monitoring system, providing historical performance graphs based on standard operating system metrics, the same metrics printed by mpstat(1), iostat(1), and system utilities. Sumit logs in to AcmeMon to see for himself.

Sumit begins with a methodology called the USE method (defined in Chapter 2, Methodologies, Section 2.5.9) to quickly check for resource bottlenecks. As the database team reported, utilization for the disks is high, around 80%, while for the other resources (CPU, network) utilization is much lower. The historical data shows that disk utilization has been steadily increasing during the past week, while CPU utilization has been steady. AcmeMon doesn't provide saturation or error statistics for the disks, so to complete the USE method Sumit must log in to the server and run some commands.

He checks disk error counters from /sys; they are zero. He runs iostat(1) with an interval of one second and watches utilization and saturation metrics over time. AcmeMon reported 80% utilization but uses a one-minute interval. At one-second granularity, Sumit can see that disk utilization fluctuates, often hitting 100% and causing levels of saturation and increased disk I/O latency.

To further confirm that this is blocking the database—and isn't asynchronous with respect to the database queries—he uses a BCC/BPF tracing tool called offcputime(8) to capture stack traces whenever the database was descheduled by the kernel, along with the time spent off-CPU. The stack traces show that the database is often blocking during a file system read, during a query. This is enough evidence for Sumit.

The next question is why. The disk performance statistics appear to be consistent with high load. Sumit performs workload characterization to understand this further, using iostat(1) to measure IOPS, throughput, average disk I/O latency, and the read/write ratio. For more details, Sumit can use disk I/O tracing; however, he is satisfied that this already points to a case of high disk load, and not a problem with the disks.

Sumit adds more details to the ticket, stating what he checked and including screenshots of the commands used to study the disks. His summary so far is that the disks are under high load, which increases I/O latency and is slowing the queries. However, the disks appear to be acting normally for the load. He asks if there is a simple explanation: did the database load increase?

The database team responds that it did not, and that the rate of queries (which isn't reported by AcmeMon) has been steady. This sounds consistent with an earlier finding, that CPU utilization was also steady.

Sumit thinks about what else could cause higher disk I/O load without a noticeable increase in CPU and has a quick talk with his colleagues about it. One of them suggests file system fragmentation, which is expected when the file system approaches 100% capacity. Sumit finds that it is only at 30%.

Sumit knows he can perform drill-down analysis[7] to understand the exact causes of disk I/O, but this can be time-consuming. He tries to think of other easy explanations that he can check quickly first, based on his knowledge of the kernel I/O stack. He remembers that this disk I/O is largely caused by file system cache (page cache) misses.

Sumit checks the file system cache hit ratio using cachestat(8)[8] and finds it is currently at 91%. This sounds high (good), but he has no historical data to compare it to. He logs in to other database servers that serve similar workloads and finds their cache hit ratio to be over 98%. He also finds that the file system cache size is much larger on the other servers.

Turning his attention to the file system cache size and server memory usage, he finds something that had been overlooked: a development project has a prototype application that is consuming a growing amount of memory, even though it isn't under production load yet. This memory is taken from what is available for the file system cache, reducing its hit rate and causing more file system reads to become disk reads.

---

[7] This is covered in Chapter 2, Methodologies, Section 2.5.12, Drill-Down Analysis.

[8] A BCC tracing tool covered in Chapter 8, File Systems, Section 8.6.12, cachestat.

Sumit contacts the application development team and asks them to shut down the application and move it to a different server, referring to the database issue. After they do this, Sumit watches disk utilization creep downward in AcmeMon as the file system cache recovers to its original size. The slow queries return to zero, and he closes the ticket as resolved.

1.11.2 Software Change

Pamela is a performance and scalability engineer at a small company where she works on all performance-related activities. The application developers have developed a new core feature and are unsure whether its introduction could hurt performance. Pamela decides to perform non-regression testing[9] of the new application version, before it is deployed in production.

Pamela acquires an idle server for the purpose of testing and searches for a client workload simulator. The application team had written one a while ago, although it has various limitations and known bugs. She decides to try it but wants to confirm that it adequately resembles the current production workload.

She configures the server to match the current deployment configuration and runs the client workload simulator from a different system to the server. The client workload can be characterized by studying an access log, and there is already a company tool to do this, which she uses. She also runs the tool on a production server log for different times of day and compares workloads. It appears that the client simulator applies an average production workload but doesn't account for variance. She notes this and continues her analysis.

Pamela knows a number of approaches to use at this point. She picks the easiest: increasing load from the client simulator until a limit is reached (this is sometimes called *stress testing*). The client simulator can be configured to execute a target number of client requests per second, with a default of 1,000 that she had used earlier. She decides to increase load starting at 100 and adding increments of 100 until a limit is reached, each level being tested for one minute. She writes a shell script to perform the test, which collects results in a file for plotting by other tools.

With the load running, she performs active benchmarking to determine what the limiting factors are. The server resources and server threads seem largely idle. The client simulator shows that the request throughput levels off at around 700 per second.

She switches to the new software version and repeats the test. This also reaches the 700 mark and levels off. She also analyzes the server to look for limiting factors but again cannot see any.

She plots the results, showing completed request rate versus load, to visually identify the scalability profile. Both appear to reach an abrupt ceiling.

While it appears that the software versions have similar performance characteristics, Pamela is disappointed that she wasn't able to identify the limiting factor causing the scalability ceiling. She knows she checked only server resources, and the limiter could instead be an application logic issue. It could also be elsewhere: the network or the client simulator.

[9] Some call it *regression* testing, but it is an activity intended to confirm that a software or hardware change does not cause performance to regress, hence, *non*-regression testing.

Pamela wonders if a different approach may be needed, such as running a fixed rate of operations and then characterizing resource usage (CPU, disk I/O, network I/O), so that it can be expressed in terms of a single client request. She runs the simulator at a rate of 700 per second for the current and new software and measures resource consumption. The current software drove the 32 CPUs to an average of 20% utilization for the given load. The new software drove the same CPUs to 30% utilization, for the same load. It would appear that this is indeed a regression, one that consumes more CPU resources.

Curious to understand the 700 limit, Pamela launches a higher load and then investigates all components in the data path, including the network, the client system, and the client workload generator. She also performs drill-down analysis of the server and client software. She documents what she has checked, including screenshots, for reference.

To investigate the client software she performs thread state analysis and finds that it is single-threaded! That one thread is spending 100% of its time executing on-CPU. This convinces her that this is the limiter of the test.

As an experiment, she launches the client software in parallel on different client systems. In this way, she drives the server to 100% CPU utilization for both the current and new software. The current version reaches 3,500 requests/sec, and the new version 2,300 requests/sec, consistent with earlier findings of resource consumption.

Pamela informs the application developers that there is a regression with the new software version, and she begins to profile its CPU usage using a CPU flame graph to understand why: what code paths are contributing. She notes that an average production workload was tested and that varied workloads were not. She also files a bug to note that the client workload generator is single-threaded, which can become a bottleneck.

1.11.3 More Reading

A more detailed case study is provided as Chapter 16, Case Study, which documents how I resolved a particular cloud performance issue. The next chapter introduces the methodologies used for performance analysis, and the remaining chapters cover the necessary background and specifics.

1.12 References

[Hollingsworth 94] Hollingsworth, J., Miller, B., and Cargille, J., "Dynamic Program Instrumentation for Scalable Performance Tools," *Scalable High-Performance Computing Conference (SHPCC)*, May 1994.

[Tamches 99] Tamches, A., and Miller, B., "Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels," *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, February 1999.

[Kleen 08] Kleen, A., "On Submitting Kernel Patches," *Intel Open Source Technology Center*, http://halobates.de/on-submitting-patches.pdf, 2008.

[Gregg 11a] Gregg, B., and Mauro, J., *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*, Prentice Hall, 2011.

[Gregg 15a] Gregg, B., "Linux Performance Analysis in 60,000 Milliseconds," *Netflix Technology Blog*, http://techblog.netflix.com/2015/11/linux-performance-analysis-in-60s.html, 2015.

[Dekker 18] Dekker, S., *Drift into Failure: From Hunting Broken Components to Understanding Complex Systems*, CRC Press, 2018.

[Gregg 19] Gregg, B., *BPF Performance Tools: Linux System and Application Observability*, Addison-Wesley, 2019.

[Corry 20] Corry, A., *Retrospectives Antipatterns*, Addison-Wesley, 2020.

Chapter 2

Methodologies

Give a man a fish and you feed him for a day.
Teach a man to fish and you feed him for a lifetime.
Chinese proverb (English equivalent)

I began my tech career as a junior system administrator, and I thought I could learn performance by studying command-line tools and metrics alone. I was wrong. I read man pages from top to bottom and learned the definitions for page faults, context switches, and various other system metrics, but I didn't know what to do with them: how to move from signals to solutions.

I noticed that, whenever there was a performance issue, the senior system administrators had their own mental procedures for moving quickly through tools and metrics to find the root cause. They understood which metrics were important and when they pointed to an issue, and how to use them to narrow down an investigation. It was this *know-how* that was missing from the man pages—it was typically learned by watching over the shoulder of a senior admin or engineer.

Since then I've collected, documented, shared, and developed performance *methodologies* of my own. This chapter includes these methodologies and other essential background for systems performance: concepts, terminology, statistics, and visualizations. This covers theory before later chapters dive into implementation.

The learning objectives of this chapter are:

- Understand key performance metrics: latency, utilization, and saturation.
- Develop a sense for the scale of measured time, down to nanoseconds.
- Learn tuning trade-offs, targets, and when to stop analysis.
- Identify problems of workload versus architecture.
- Consider resource versus workload analysis.
- Follow different performance methodologies, including: the USE method, workload characterization, latency analysis, static performance tuning, and performance mantras.
- Understand the basics of statistics and queueing theory.

Of all the chapters in this book, this one has changed the least since the first edition. Software, hardware, performance tools, and performance tunables have all changed over the course of my career. What have remained the same are the theory and methodologies: the durable skills covered in this chapter.

This chapter has three parts:

- **Background** introduces terminology, basic models, key performance concepts, and perspectives. Much of this will be assumed knowledge for the rest of this book.
- **Methodology** discusses performance analysis methodologies, both observational and experimental; modeling; and capacity planning.
- **Metrics** introduces performance statistics, monitoring, and visualizations.

Many of the methodologies introduced here are explored in more detail in later chapters, including the methodology sections in Chapters 5 through 10.

2.1 Terminology

The following are key terms for systems performance. Later chapters provide additional terms and describe some of these in different contexts.

- **IOPS**: Input/output operations per second is a measure of the rate of data transfer operations. For disk I/O, IOPS refers to reads and writes per second.
- **Throughput**: The rate of work performed. Especially in communications, the term is used to refer to the *data rate* (bytes per second or bits per second). In some contexts (e.g., databases) throughput can refer to the *operation rate* (operations per second or transactions per second).
- **Response time**: The time for an operation to complete. This includes any time spent waiting and time spent being serviced (*service time*), including the time to transfer the result.
- **Latency**: A measure of time an operation spends waiting to be serviced. In some contexts it can refer to the entire time for an operation, equivalent to response time. See Section 2.3, Concepts, for examples.
- **Utilization**: For resources that service requests, utilization is a measure of how busy a resource is, based on how much time in a given interval it was actively performing work. For resources that provide storage, utilization may refer to the capacity that is consumed (e.g., memory utilization).
- **Saturation**: The degree to which a resource has queued work it cannot service.
- **Bottleneck**: In systems performance, a bottleneck is a resource that limits the performance of the system. Identifying and removing systemic bottlenecks is a key activity of systems performance.
- **Workload**: The input to the system or the load applied is the workload. For a database, the workload consists of the database queries and commands sent by the clients.

- **Cache**: A fast storage area that can duplicate or buffer a limited amount of data, to avoid communicating directly with a slower tier of storage, thereby improving performance. For economic reasons, a cache is often smaller than the slower tier.

The Glossary includes more terminology for reference if needed.

2.2 Models

The following simple models illustrate some basic principles of system performance.

2.2.1 System Under Test

The performance of a system under test (SUT) is shown in Figure 2.1.

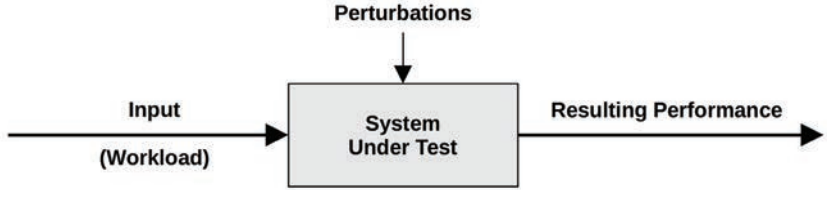


Figure 2.1 Block diagram of system under test

It is important to be aware that perturbations (interference) can affect results, including those caused by scheduled system activity, other users of the system, and other workloads. The origin of the perturbations may not be obvious, and careful study of system performance may be required to determine it. This can be particularly difficult in some cloud environments, where other activity (by guest tenants) on the physical host system may not be observable from within a guest SUT.

Another difficulty with modern environments is that they may be composed of several networked components servicing the input workload, including load balancers, proxy servers, web servers, caching servers, application servers, database servers, and storage systems. The mere act of mapping the environment may help to reveal previously overlooked sources of perturbations. The environment may also be modeled as a network of queueing systems, for analytical study.

2.2.2 Queueing System

Some components and resources can be modeled as a queueing system so that their performance under different situations can be predicted based on the model. Disks are commonly modeled as a queueing system, which can predict how response time degrades under load. Figure 2.2 shows a simple queueing system.

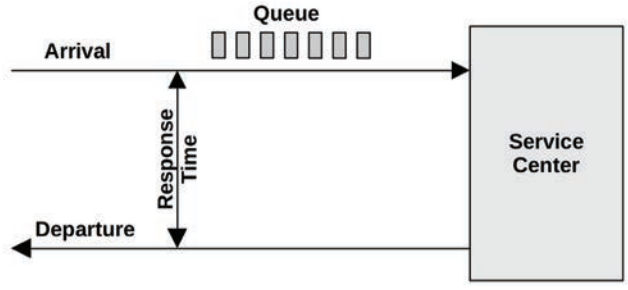


Figure 2.2 Simple queueing model

The field of queueing theory, introduced in Section 2.6, Modeling, studies queueing systems and networks of queueing systems.

2.3 Concepts

The following are important concepts of systems performance and are assumed knowledge for the rest of this chapter and this book. The topics are described in a generic manner, before implementation-specific details are introduced in the Architecture sections of later chapters.

2.3.1 Latency

For some environments, latency is the sole focus of performance. For others, it is the top one or two key metrics for analysis, along with throughput.

As an example of latency, Figure 2.3 shows a network transfer, such as an HTTP GET request, with the time split into latency and data transfer components.

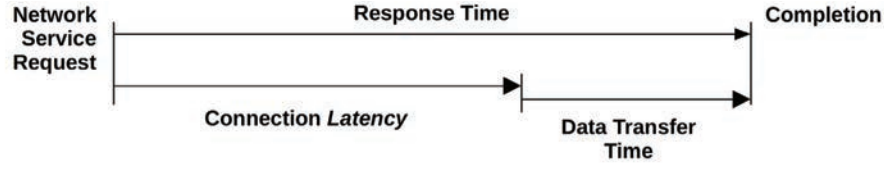


Figure 2.3 Network connection latency

The latency is the time spent waiting before an operation is performed. In this example, the operation is a network service request to transfer data. Before this operation can take place, the system must wait for a network connection to be established, which is latency for this operation. The *response time* spans this latency and the operation time.

Because latency can be measured from different locations, it is often expressed with the target of the measurement. For example, the load time for a website may be composed of three different times measured from different locations: *DNS latency*, *TCP connection latency*, and then *TCP data*

transfer time. DNS latency refers to the entire DNS operation. TCP connection latency refers to the initialization only (TCP handshake).

At a higher level, all of these, including the TCP data transfer time, may be treated as latency of something else. For example, the time from when the user clicks a website link to when the resulting page is fully loaded may be termed *latency*, which includes the time for the browser to fetch a web page over a network and render it. Since the single word "latency" can be ambiguous, it is best to include qualifying terms to explain what it measures: request latency, TCP connection latency, etc.

As latency is a time-based metric, various calculations are possible. Performance issues can be quantified using latency and then ranked because they are expressed using the same units (time). Predicted speedup can also be calculated, by considering when latency can be reduced or removed. Neither of these can be accurately performed using an IOPS metric, for example.

For reference, time orders of magnitude and their abbreviations are listed in Table 2.1.

Table 2.1 **Units of time**

Unit	Abbreviation	Fraction of 1 Second
Minute	m	60
Second	s	1
Millisecond	ms	0.001 or 1/1000 or 1×10^{-3}
Microsecond	µs	0.000001 or 1/1000000 or 1×10^{-6}
Nanosecond	ns	0.000000001 or 1/1000000000 or 1×10^{-9}
Picosecond	ps	0.000000000001 or 1/1000000000000 or 1×10^{-12}

When possible, converting other metric types to latency or time allows them to be compared. If you had to choose between 100 network I/O or 50 disk I/O, how would you know which would perform better? It's a complicated question, involving many factors: network hops, rate of network drops and retransmits, I/O size, random or sequential I/O, disk types, and so on. But if you compare 100 ms of total network I/O and 50 ms of total disk I/O, the difference is clear.

2.3.2 Time Scales

While times can be compared numerically, it also helps to have an instinct about time, and reasonable expectations for latency from different sources. System components operate over vastly different time scales (orders of magnitude), to the extent that it can be difficult to grasp just how big those differences are. In Table 2.2, example latencies are provided, starting with CPU register access for a 3.5 GHz processor. To demonstrate the differences in time scales we're working with, the table shows an average time that each operation might take, scaled to an imaginary system in which a CPU cycle—0.3 ns (about one-third of one-billionth[1] of a second) in real life—takes one full second.

[1] US billionth: 1/1000,000,000

Table 2.2 **Example time scale of system latencies**

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	3 ns	10 s
Level 3 cache access	10 ns	33 s
Main memory access (DRAM, from CPU)	100 ns	6 min
Solid-state disk I/O (flash memory)	10–100 µs	9–90 hours
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Lightweight hardware virtualization boot	100 ms	11 years
Internet: San Francisco to Australia	183 ms	19 years
OS virtualization system boot	< 1 s	105 years
TCP timer-based retransmit	1–3 s	105–317 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system boot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

As you can see, the time scale for CPU cycles is tiny. The time it takes light to travel 0.5 m, perhaps the distance from your eyes to this page, is about 1.7 ns. During the same time, a modern CPU may have executed five CPU cycles and processed several instructions.

For more about CPU cycles and latency, see Chapter 6, CPUs, and for disk I/O latency, Chapter 9, Disks. The Internet latencies included are from Chapter 10, Network, which has more examples.

2.3.3 Trade-Offs

You should be aware of some common performance trade-offs. The good/fast/cheap "pick two" trade-off is shown in Figure 2.4, alongside the terminology adjusted for IT projects.

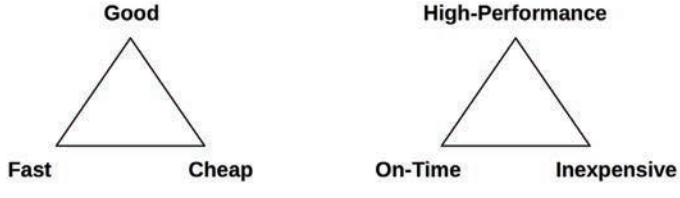


Figure 2.4 Trade-offs: pick two

Many IT projects choose on-time and inexpensive, leaving performance to be fixed later. This choice can become problematic when earlier decisions inhibit improving performance, such as choosing and populating a suboptimal storage architecture, using a programming language or operating system that is implemented inefficiently, or selecting a component that lacks comprehensive performance analysis tools.

A common trade-off in performance tuning is the one between CPU and memory, as memory can be used to cache results, reducing CPU usage. On modern systems with an abundance of CPU, the trade may work the other way: CPU time may be spent compressing data to reduce memory usage.

Tunable parameters often come with trade-offs. Here are a couple of examples:

- **File system record size** (or block size): Small record sizes, close to the application I/O size, will perform better for random I/O workloads and make more efficient use of the file system cache while the application is running. Large record sizes will improve streaming workloads, including file system backups.
- **Network buffer size**: Small buffer sizes will reduce the memory overhead per connection, helping the system scale. Large sizes will improve network throughput.

Look for such trade-offs when making changes to the system.

2.3.4 Tuning Efforts

Performance tuning is most effective when done closest to where the work is performed. For workloads driven by applications, this means within the application itself. Table 2.3 shows an example software stack with tuning possibilities.

By tuning at the application level, you may be able to eliminate or reduce database queries and improve performance by a large factor (e.g., 20x). Tuning down to the storage device level may eliminate or improve storage I/O, but a tax has already been paid in executing higher-level OS stack code, so this may improve resulting application performance only by percentages (e.g., 20%).

Table 2.3 **Example targets of tuning**

Layer	Example Tuning Targets
Application	Application logic, request queue sizes, database queries performed
Database	Database table layout, indexes, buffering
System calls	Memory-mapped or read/write, sync or async I/O flags
File system	Record size, cache size, file system tunables, journaling
Storage	RAID level, number and type of disks, storage tunables

There is another reason for finding large performance wins at the application level. Many of today's environments target rapid deployment for features and functionality, pushing software

changes into production weekly or daily.[2] Application development and testing therefore tend to focus on correctness, leaving little or no time for performance measurement or optimization before production deployment. These activities are conducted later, when performance becomes a problem.

While the application can be the most effective level at which to tune, it isn't necessarily the most effective level from which to base observation. Slow queries may be best understood from their time spent on-CPU, or from the file system and disk I/O that they perform. These are observable from operating system tools.

In many environments (especially cloud computing) the application level is under constant development, pushing software changes into production weekly or daily. Large performance wins, including fixes for regressions, are frequently found as the application code changes. In these environments, tuning for the operating system and observability from the operating system can be easy to overlook. Remember that operating system performance analysis can also identify application-level issues, not just OS-level issues, in some cases more easily than from the application alone.

2.3.5 Level of Appropriateness

Different organizations and environments have different requirements for performance. You may have joined an organization where it is the norm to analyze much deeper than you've seen before, or even knew was possible. Or you may find that, in your new workplace, what you consider basic analysis is considered advanced and has never before been performed (good news: low-hanging fruit!).

This doesn't necessarily mean that some organizations are doing it right and some wrong. It depends on the return on investment (ROI) for performance expertise. Organizations with large data centers or large cloud environments may employ a team of performance engineers who analyze everything, including kernel internals and CPU performance counters, and make frequent use of a variety of tracing tools. They may also formally model performance and develop accurate predictions for future growth. For environments spending millions per year on computing, it can be easy to justify hiring such a performance team, as the wins they find are the ROI. Small startups with modest computing spend may only perform superficial checks, trusting third-party monitoring solutions to check their performance and provide alerts.

However, as introduced in Chapter 1, systems performance is not just about cost: it is also about the end-user experience. A startup may find it necessary to invest in performance engineering to improve website or application latency. The ROI here is not necessarily a reduction in cost, but happier customers instead of ex-customers.

The most extreme environments include stock exchanges and high-frequency traders, where performance and latency are critical and can justify intense effort and expense. As an example of this, a transatlantic cable between the New York and London exchanges was planned with a cost of $300 million, to reduce transmission latency by 6 ms [Williams 11].

[2] Examples of environments that change rapidly include the Netflix cloud and Shopify, which push multiple changes per day.

When doing performance analysis, the level of appropriateness also comes in to play in deciding when to stop analysis.

2.3.6 When to Stop Analysis

A challenge whenever doing performance analysis is knowing when to stop. There are so many tools, and so many things to examine!

When I teach performance classes (as I've begun to do again recently), I can give my students a performance issue that has three contributing reasons, and find that some students stop after finding one reason, others two, and others all three. Some students keep going, trying to find even more reasons for the performance issue. Who is doing it right? It might be easy to say you should stop after finding all three reasons, but for real-life issues you don't know the number of causes.

Here are three scenarios where you may consider stopping analysis, with some personal examples:

- **When you've explained the bulk of the performance problem**. A Java application was consuming three times more CPU than it had been. The first issue I found was one of exception stacks consuming CPU. I then quantified time in those stacks and found that they accounted for only 12% of the overall CPU footprint. If that figure had been closer to 66%, I could have stopped analysis—the 3x slowdown would have been accounted for. But in this case, at 12%, I needed to keep looking.
- **When the potential ROI is less than the cost of analysis**. Some performance issues I work on can deliver wins measured in tens of millions of dollars per year. For these I can justify spending months of my own time (engineering cost) on analysis. Other performance wins, say for tiny microservices, may be measured in hundreds of dollars: it may not be worth even an hour of engineering time to analyze them. Exceptions might include when I have nothing better to do with company time (which never happens in practice) or if I suspected that this might be a canary for a bigger issue later on, and therefore worth debugging before the problem grows.
- **When there are bigger ROIs elsewhere**. Even if the previous two scenarios have not been met, there may be larger ROIs elsewhere that take priority.

If you are working full-time as a performance engineer, prioritizing the analysis of different issues based on their potential ROI is likely a daily task.

2.3.7 Point-in-Time Recommendations

The performance characteristics of environments change over time, due to the addition of more users, newer hardware, and updated software or firmware. An environment currently limited by a 10 Gbit/s network infrastructure may start to experience a bottleneck in disk or CPU performance after an upgrade to 100 Gbits/s.

Performance recommendations, especially the values of tunable parameters, are valid only at a specific *point in time*. What may have been the best advice from a performance expert one week may become invalid a week later after a software or hardware upgrade, or after adding more users.

Tunable parameter values found by searching on the Internet can provide quick wins—in *some* cases. They can also cripple performance if they are not appropriate for your system or workload, were appropriate once but are not now, or are appropriate only as a temporary workaround for a software bug that is fixed properly in a later software upgrade. It is akin to raiding someone else's medicine cabinet and taking drugs that may not be appropriate for you, may have expired, or were supposed to be taken only for a short duration.

It can be useful to browse such recommendations just to see which tunable parameters exist and have needed changing in the past. Your task then becomes to see whether and how these should be tuned for your system and workload. But you may still miss an important parameter if others have not needed to tune that one before, or have tuned it but haven't shared their experience anywhere.

When changing tunable parameters, it can be helpful to store them in a version control system with a detailed history. (You may already do something similar when using configuration management tools such as Puppet, Salt, Chef, etc.) That way the times and reasons that tunables were changed can be examined later on.

2.3.8 Load vs. Architecture

An application can perform badly due to an issue with the software configuration and hardware on which it is running: its architecture and implementation. However, an application can also perform badly simply due to too much load being applied, resulting in queueing and long latencies. Load and architecture are pictured in Figure 2.5.

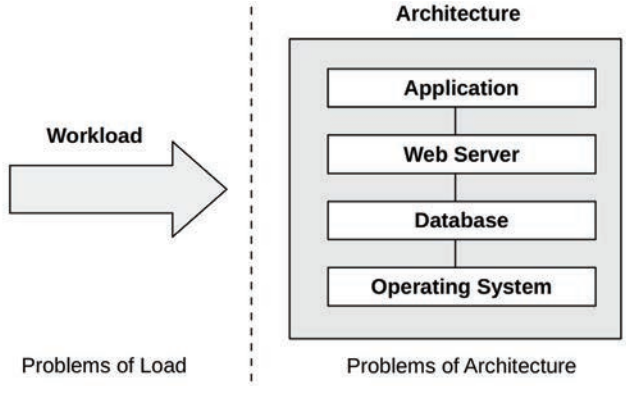


Figure 2.5 Load versus architecture

If analysis of the architecture shows queueing of work but no problems with how the work is performed, the issue may be too much load applied. In a cloud computing environment, this is the point where more server instances can be introduced on demand to handle the work.

For example, an issue of architecture may be a single-threaded application that is busy on-CPU, with requests queueing while other CPUs are available and idle. In this case, performance is

limited by the application's single-threaded architecture. Another issue of architecture may be a multi-threaded program that contends for a single lock, such that only one thread can make forward progress while others wait.

An issue of load may be a multithreaded application that is busy on all available CPUs, with requests still queueing. In this case, performance is limited by the available CPU capacity, or put differently, by there being more load than the CPUs can handle.

2.3.9 Scalability

The performance of the system under increasing load is its *scalability*. Figure 2.6 shows a typical throughput profile as a system's load increases.

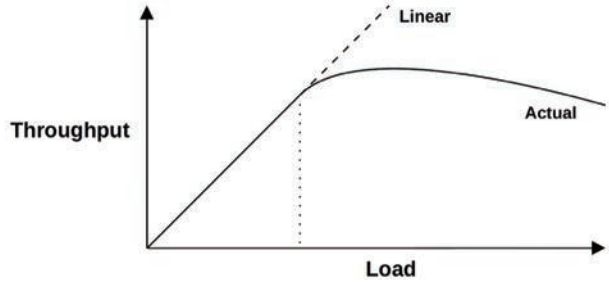


Figure 2.6 Throughput versus load

For some period, linear scalability is observed. A point is then reached, marked with a dotted line, where contention for a resource begins to degrade throughput. This point can be described as a *knee point*, as it is the boundary between two functions. Beyond this point, the throughput profile departs from linear scalability, as contention for the resource increases. Eventually the overheads for increased contention and coherency cause less work to be completed and throughput to decrease.

This point may occur when a component reaches 100% utilization: the *saturation point*. It may also occur when a component approaches 100% utilization and queueing begins to be frequent and significant.

An example system that may exhibit this profile is an application that performs heavy computation, with more load added as additional threads. As the CPUs approach 100% utilization, response time begins to degrade as CPU scheduler latency increases. After peak performance, at 100% utilization, throughput begins to decrease as more threads are added, causing more context switches, which consume CPU resources and cause less actual work to be completed.

The same curve can be seen if you replace "load" on the x-axis with a resource such as CPU cores. For more on this topic, see Section 2.6, Modeling.

The degradation of performance for nonlinear scalability, in terms of average response time or latency, is graphed in Figure 2.7 [Cockcroft 95].

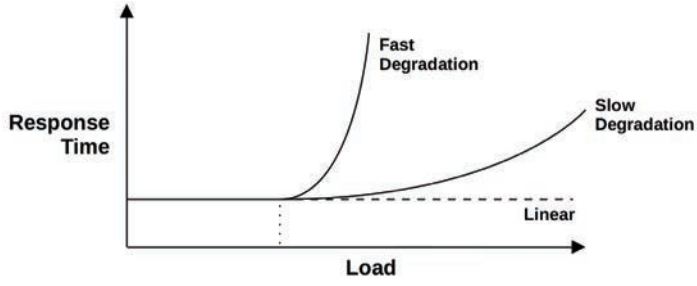


Figure 2.7 Performance degradation

Higher response time is, of course, bad. The "fast" degradation profile may occur for memory load, when the system begins moving memory pages to disk to free main memory. The "slow" degradation profile may occur for CPU load.

Another "fast" profile example is disk I/O. As load (and the resulting disk utilization) increases, I/O becomes more likely to queue behind other I/O. An idle rotational (not solid state) disk may serve I/O with a response time of about 1 ms, but when load increases, this can approach 10 ms. This is modeled in Section 2.6.5, Queueing Theory, under M/D/1 and 60% Utilization, and disk performance is covered in Chapter 9, Disks.

Linear scalability of response time could occur if the application begins to return errors when resources are unavailable, instead of queueing work. For example, a web server may return 503 "Service Unavailable" instead of adding requests to a queue, so that those requests that are served can be performed with a consistent response time.

2.3.10 Metrics

Performance metrics are selected statistics generated by the system, applications, or additional tools that measure activity of interest. They are studied for performance analysis and monitoring, either numerically at the command line or graphically using visualizations.

Common types of system performance metrics include:

- **Throughput**: Either operations or data volume per second
- **IOPS**: I/O operations per second
- **Utilization**: How busy a resource is, as a percentage
- **Latency**: Operation time, as an average or percentile

The usage of throughput depends on its context. Database throughput is usually a measure of queries or requests (operations) per second. Network throughput is a measure of bits or bytes (volume) per second.

IOPS is a throughput measurement for I/O operations only (reads and writes). Again, context matters, and definitions can vary.

Overhead

Performance metrics are not free; at some point, CPU cycles must be spent to gather and store them. This causes overhead, which can negatively affect the performance of the target of measurement. This is called the *observer effect*. (It is often confused with Heisenberg's Uncertainty Principle, which describes the limit of precision at which pairs of physical properties, such as position and momentum, may be known.)

Issues

You might assume that a software vendor has provided metrics that are well chosen, are bug-free, and provide complete visibility. In reality, metrics can be confusing, complicated, unreliable, inaccurate, and even plain wrong (due to bugs). Sometimes a metric was correct in one software version but did not get updated to reflect the addition of new code and code paths.

For more about problems with metrics, see Chapter 4, Observability Tools, Section 4.6, Observing Observability.

2.3.11 Utilization

The term *utilization*[3] is often used for operating systems to describe device usage, such as for the CPU and disk devices. Utilization can be time-based or capacity-based.

Time-Based

Time-based utilization is formally defined in queueing theory. For example [Gunther 97]:

> the average amount of time the server or resource was busy

along with the ratio

$$U = B/T$$

where U = utilization, B = total time the system was busy during T, the observation period.

This is also the "utilization" most readily available from operating system performance tools. The disk monitoring tool iostat(1) calls this metric `%b` for *percent busy*, a term that better conveys the underlying metric: B/T.

This utilization metric tells us how busy a component is: when a component approaches 100% utilization, performance can seriously degrade when there is contention for the resource. Other metrics can be checked to confirm and to see if the component has therefore become a system bottleneck.

Some components can service multiple operations in parallel. For them, performance may not degrade much at 100% utilization as they can accept more work. To understand this, consider a

[3] Spelled *utilisation* in some parts of the world.

building elevator. It may be considered utilized when it is moving between floors, and not utilized when it is idle waiting. However, the elevator may be able to accept more passengers even when it is busy 100% of the time responding to calls—that is, it is at 100% utilization.

A disk that is 100% busy may also be able to accept and process more work, for example, by buffering writes in the on-disk cache to be completed later. Storage arrays frequently run at 100% utilization because *some* disk is busy 100% of the time, but the array has plenty of idle disks and can accept more work.

Capacity-Based

The other definition of utilization is used by IT professionals in the context of capacity planning [Wong 97]:

> A system or component (such as a disk drive) is able to deliver a certain amount of throughput. At any level of performance, the system or component is working at some proportion of its capacity. That proportion is called the utilization.

This defines utilization in terms of capacity instead of time. It implies that a disk at 100% utilization *cannot* accept any more work. With the time-based definition, 100% utilization only means it is busy 100% of the time.

> 100% busy does not mean 100% capacity.

For the elevator example, 100% capacity may mean the elevator is at its maximum payload capacity and cannot accept more passengers.

In an ideal world, we would be able to measure both types of utilization for a device, so that, for example, you would know when a disk is 100% busy and performance begins to degrade due to contention, and also when it is at 100% capacity and cannot accept more work. Unfortunately, this usually isn't possible. For a disk, it would require knowledge of what the disk's on-board controller was doing and a prediction of capacity. Disks do not currently provide this information.

In this book, *utilization* usually refers to the time-based version, which you could also call *non-idle time*. The capacity version is used for some volume-based metrics, such as memory usage.

2.3.12 Saturation

The degree to which more work is requested of a resource than it can process is *saturation*. Saturation begins to occur at 100% utilization (capacity-based), as extra work cannot be processed and begins to queue. This is pictured in Figure 2.8.

The figure pictures saturation increasing linearly beyond the 100% capacity-based utilization mark as load continues to increase. Any degree of saturation is a performance issue, as time is spent waiting (latency). For time-based utilization (percent busy), queueing and therefore saturation may not begin at the 100% utilization mark, depending on the degree to which the resource can operate on work in parallel.

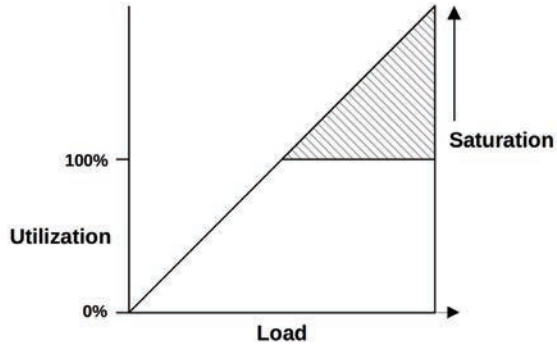


Figure 2.8 Utilization versus saturation

2.3.13 Profiling

Profiling builds a picture of a target that can be studied and understood. In the field of computing performance, profiling is typically performed by *sampling* the state of the system at timed intervals and then studying the set of samples.

Unlike the previous metrics covered, including IOPS and throughput, the use of sampling provides a *coarse* view of the target's activity. How coarse depends on the rate of sampling.

As an example of profiling, CPU usage can be understood in reasonable detail by sampling the CPU instruction pointer or stack trace at frequent intervals to gather statistics on the code paths that are consuming CPU resources. This topic is covered in Chapter 6, CPUs.

2.3.14 Caching

Caching is frequently used to improve performance. A cache stores results from a slower storage tier in a faster storage tier, for reference. An example is caching disk blocks in main memory (RAM).

Multiple tiers of caches may be used. CPUs commonly employ multiple hardware caches for main memory (Levels 1, 2, and 3), beginning with a very fast but small cache (Level 1) and increasing in both storage size and access latency. This is an economic trade-off between density and latency; levels and sizes are chosen for the best performance for the on-chip space available. These caches are covered in Chapter 6, CPUs.

There are many other caches present in a system, many of them implemented in software using main memory for storage. See Chapter 3, Operating Systems, Section 3.2.11, Caching, for a list of caching layers.

One metric for understanding cache performance is each cache's *hit ratio*—the number of times the needed data was found in the cache (hits) versus the total accesses (hits + misses):

```
hit ratio = hits / (hits + misses)
```

The higher, the better, as a higher ratio reflects more data successfully accessed from faster media. Figure 2.9 shows the expected performance improvement for increasing cache hit ratios.

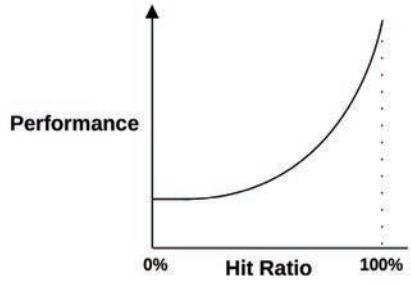


Figure 2.9 Cache hit ratio and performance

The performance difference between 98% and 99% is much greater than that between 10% and 11%. This is a nonlinear profile because of the difference in speed between cache hits and misses—the two storage tiers at play. The greater the difference, the steeper the slope becomes.

Another metric for understanding cache performance is the *cache miss rate*, in terms of misses per second. This is proportional (linear) to the performance penalty of each miss and can be easier to interpret.

For example, workloads A and B perform the same task using different algorithms and use a main memory cache to avoid reading from disk. Workload A has a cache hit ratio of 90%, and workload B has a cache hit ratio of 80%. This information alone suggests workload A performs better. What if workload A had a miss rate of 200/s and workload B, 20/s? In those terms, workload B performs 10x *fewer* disk reads, which may complete the task much sooner than A. To be certain, the total runtime for each workload can be calculated as

$$\text{runtime} = (\text{hit rate} \times \text{hit latency}) + (\text{miss rate} \times \text{miss latency})$$

This calculation uses the average hit and miss latencies and assumes the work is serialized.

Algorithms

Cache management algorithms and policies determine what to store in the limited space available for a cache.

Most recently used (MRU) refers to a cache *retention policy*, which decides what to favor keeping in the cache: the objects that have been used most recently. *Least recently used* (LRU) can refer to an equivalent cache *eviction policy*, deciding what objects to remove from the cache when more space is needed. There are also *most frequently used* (MFU) and *least frequently used* (LFU) policies.

You may encounter *not frequently used* (NFU), which may be an inexpensive but less thorough version of LRU.

Hot, Cold, and Warm Caches

These words are commonly used to describe the state of the cache:

- **Cold**: A *cold cache* is empty, or populated with unwanted data. The hit ratio for a cold cache is zero (or near zero as it begins to warm up).

- **Warm**: A *warm cache* is one that is populated with useful data but doesn't have a high enough hit ratio to be considered hot.
- **Hot**: A *hot cache* is populated with commonly requested data and has a high hit ratio, for example, over 99%.
- **Warmth**: Cache warmth describes how hot or cold a cache is. An activity that improves cache warmth is one that aims to improve the cache hit ratio.

When caches are first initialized, they begin cold and then warm up over time. When the cache is large or the next-level storage is slow (or both), the cache can take a long time to become populated and warm.

For example, I worked on a storage appliance that had 128 Gbytes of DRAM as a file system cache, 600 Gbytes of flash memory as a second-level cache, and rotational disks for storage. With a random read workload, the disks delivered around 2,000 reads/s. With an 8 Kbyte I/O size, this meant that the caches could warm up at a rate of only 16 Mbytes/s (2,000 × 8 Kbytes). When both caches began cold, it took more than 2 hours for the DRAM cache to warm up, and more than 10 hours for the flash memory cache to warm up.

2.3.15 Known-Unknowns

Introduced in the Preface, the notion of *known-knowns*, *known-unknowns*, and *unknown-unknowns* is important for the field of performance. The breakdown is as follows, with examples for systems performance analysis:

- **Known-knowns**: These are things you know. You know you should be checking a performance metric, and you know its current value. For example, you know you should be checking CPU utilization, and you also know that the value is 10% on average.
- **Known-unknowns**: These are things you know that you do not know. You know you can check a metric or the existence of a subsystem, but you haven't yet observed it. For example, you know you could use profiling to check what is making the CPUs busy, but have yet to do so.
- **Unknown-unknowns**: These are things you do not know that you do not know. For example, you may not know that device interrupts can become heavy CPU consumers, so you are not checking them.

Performance is a field where "the more you know, the more you don't know." The more you learn about systems, the more unknown-unknowns you become aware of, which are then known-unknowns that you can check on.

2.4 Perspectives

There are two common perspectives for performance analysis, each with different audiences, metrics, and approaches. They are *workload analysis* and *resource analysis*. They can be thought of as either top-down or bottom-up analysis of the operating system software stack, as shown in Figure 2.10.

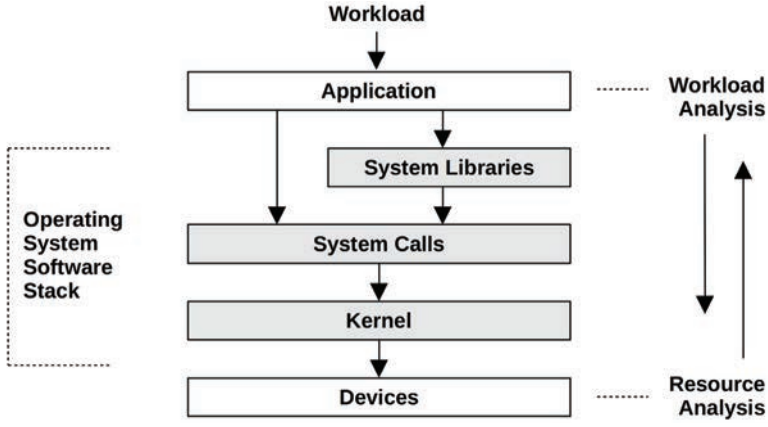


Figure 2.10 Analysis perspectives

Section 2.5, Methodology, provides specific strategies to apply for each. These perspectives are introduced here in more detail.

2.4.1 Resource Analysis

Resource analysis begins with analysis of the system resources: CPUs, memory, disks, network interfaces, buses, and interconnects. It is most likely performed by system administrators—those responsible for the physical resources. Activities include

- **Performance issue investigations**: To see if a particular type of resource is responsible
- **Capacity planning**: For information to help size new systems, and to see when existing system resources may become exhausted

This perspective focuses on utilization, to identify when resources are at or approaching their limit. Some resource types, such as CPUs, have utilization metrics readily available. Utilization for other resources can be estimated based on available metrics, for example, estimating network interface utilization by comparing the send and receive megabits per second (throughput) with the known or expected maximum bandwidth.

Metrics best suited for resource analysis include:

- IOPS
- Throughput
- Utilization
- Saturation

These measure what the resource is being asked to do, and how utilized or saturated it is for a given load. Other types of metrics, including latency, are also useful to see how well the resource is responding for the given workload.

Resource analysis is a common approach to performance analysis, in part thanks to the widely available documentation on the topic. Such documentation focuses on the operating system "stat" tools: vmstat(8), iostat(1), mpstat(1). It's important when you read such documentation to understand that this is one perspective, but not the only perspective.

2.4.2 Workload Analysis

Workload analysis (see Figure 2.11) examines the performance of applications: the workload applied and how the application is responding. It is most commonly used by application developers and support staff—those responsible for the application software and configuration.

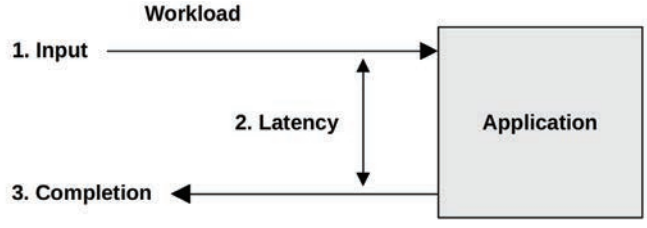


Figure 2.11 Workload analysis

The targets for workload analysis are:

- **Requests**: The workload applied
- **Latency**: The response time of the application
- **Completion**: Looking for errors

Studying workload requests typically involves checking and summarizing their attributes: this is the process of *workload characterization* (described in more detail in Section 2.5, Methodology). For databases, these attributes may include the client host, database name, tables, and query string. This data may help identify unnecessary work, or unbalanced work. Even when a system is performing its current workload well (low latency), examining these attributes may identify ways to reduce or eliminate the work applied. Keep in mind that the fastest query is the one you don't do at all.

Latency (response time) is the most important metric for expressing application performance. For a MySQL database, it's query latency; for Apache, it's HTTP request latency; and so on. In these contexts, the term *latency* is used to mean the same as response time (refer to Section 2.3.1, Latency, for more about context).

The tasks of workload analysis include identifying and confirming issues—for example, by looking for latency beyond an acceptable threshold—then finding the source of the latency and confirming that the latency is improved after applying a fix. Note that the starting point is the application. Investigating latency usually involves drilling down deeper into the application, libraries, and the operating system (kernel).

System issues may be identified by studying characteristics related to the completion of an event, including its error status. While a request may complete quickly, it may do so with an error status that causes the request to be retried, accumulating latency.

Metrics best suited for workload analysis include:

- Throughput (transactions per second)
- Latency

These measure the rate of requests and the resulting performance.

2.5 Methodology

When faced with an underperforming and complicated system environment, the first challenge can be knowing where to begin your analysis and how to proceed. As I said in Chapter 1, performance issues can arise from anywhere, including software, hardware, and any component along the data path. Methodologies can help you approach these complex systems by showing where to start your analysis and suggesting an effective procedure to follow.

This section describes many performance methodologies and procedures for system performance and tuning, some of which I developed. These methodologies help beginners get started and serve as reminders for experts. Some *anti-methodologies* have also been included.

To help summarize their role, these methodologies have been categorized as different types, such as observational analysis and experimental analysis, as shown in Table 2.4.

Table 2.4 **Generic system performance methodologies**

Section	Methodology	Type
2.5.1	Streetlight anti-method	Observational analysis
2.5.2	Random change anti-method	Experimental analysis
2.5.3	Blame-someone-else anti-method	Hypothetical analysis
2.5.4	Ad hoc checklist method	Observational and experimental analysis
2.5.5	Problem statement	Information gathering
2.5.6	Scientific method	Observational analysis
2.5.7	Diagnosis cycle	Analysis life cycle
2.5.8	Tools method	Observational analysis
2.5.9	USE method	Observational analysis
2.5.10	RED method	Observational analysis
2.5.11	Workload characterization	Observational analysis, capacity planning
2.5.12	Drill-down analysis	Observational analysis
2.5.13	Latency analysis	Observational analysis
2.5.14	Method R	Observational analysis
2.5.15	Event tracing	Observational analysis
2.5.16	Baseline statistics	Observational analysis
2.5.17	Static performance tuning	Observational analysis, capacity planning

Section	Methodology	Type
2.5.18	Cache tuning	Observational analysis, tuning
2.5.19	Micro-benchmarking	Experimental analysis
2.5.20	Performance mantras	Tuning
2.6.5	Queueing theory	Statistical analysis, capacity planning
2.7	Capacity planning	Capacity planning, tuning
2.8.1	Quantifying performance gains	Statistical analysis
2.9	Performance monitoring	Observational analysis, capacity planning

Performance monitoring, queueing theory, and capacity planning are covered later in this chapter. Other chapters also recast some of these methodologies in different contexts and provide some additional methodologies for specific targets of performance analysis. Table 2.5 lists these additional methodologies.

Table 2.5 **Additional performance methodologies**

Section	Methodology	Type
1.10.1	Linux performance analysis in 60s	Observational analysis
5.4.1	CPU profiling	Observational analysis
5.4.2	Off-CPU analysis	Observational analysis
6.5.5	Cycle analysis	Observational analysis
6.5.8	Priority tuning	Tuning
6.5.8	Resource controls	Tuning
6.5.9	CPU binding	Tuning
7.4.6	Leak detection	Observational analysis
7.4.10	Memory shrinking	Experimental analysis
8.5.1	Disk analysis	Observational analysis
8.5.7	Workload separation	Tuning
9.5.10	Scaling	Capacity planning, tuning
10.5.6	Packet sniffing	Observational analysis
10.5.7	TCP analysis	Observational analysis
12.3.1	Passive benchmarking	Experimental analysis
12.3.2	Active benchmarking	Observational analysis
12.3.6	Custom benchmarks	Software development
12.3.7	Ramping load	Experimental analysis
12.3.8	Sanity check	Observational analysis

The following sections begin with commonly used but weaker methodologies for comparison, including the anti-methodologies. For the analysis of performance issues, the first methodology you should attempt is the problem statement method, before moving on to others.

2.5.1 Streetlight Anti-Method

This method is actually the *absence* of a deliberate methodology. The user analyzes performance by choosing observability tools that are familiar, found on the Internet, or just at random to see if anything obvious shows up. This approach is hit or miss and can overlook many types of issues.

Tuning performance may be attempted in a similar trial-and-error fashion, setting whatever tunable parameters are known and familiar to different values to see if that helps.

Even when this method reveals an issue, it can be slow as tools or tunings unrelated to the issue are found and tried, just because they're familiar. This methodology is therefore named after an observational bias called the *streetlight effect*, illustrated by this parable:

> One night a police officer sees a drunk searching the ground beneath a streetlight and asks what he is looking for. The drunk says he has lost his keys. The police officer can't find them either and asks: "Are you sure you lost them here, under the streetlight?" The drunk replies: "No, but this is where the light is best."

The performance equivalent would be looking at top(1), not because it makes sense, but because the user doesn't know how to read other tools.

An issue that this methodology does find may be *an* issue but not *the* issue. Other methodologies quantify findings, so that false positives can be ruled out more quickly, and bigger issues prioritized.

2.5.2 Random Change Anti-Method

This is an experimental anti-methodology. The user randomly guesses where the problem may be and then changes things until it goes away. To determine whether performance has improved or not as a result of each change, a metric is studied, such as application runtime, operation time, latency, operation rate (operations per second), or throughput (bytes per second). The approach is as follows:

1. Pick a random item to change (e.g., a tunable parameter).
2. Change it in one direction.
3. Measure performance.
4. Change it in the other direction.
5. Measure performance.
6. Were the results in step 3 or step 5 better than the baseline? If so, keep the change and go back to step 1.

While this process may eventually unearth tuning that works for the tested workload, it is very time-consuming and can also result in tuning that doesn't make sense in the long term. For

example, an application change may improve performance because it works around a database or operating system bug that is later fixed. But the application will still have that tuning that no longer makes sense, and that no one understood properly in the first place.

Another risk is where a change that isn't properly understood causes a worse problem during peak production load, and a need to back out the change.

2.5.3 Blame-Someone-Else Anti-Method

This anti-methodology follows these steps:

1. Find a system or environment component for which you are not responsible.
2. Hypothesize that the issue is with that component.
3. Redirect the issue to the team responsible for that component.
4. When proven wrong, go back to step 1.

> "Maybe it's the network. Can you check with the network team if they've had dropped packets or something?"

Instead of investigating performance issues, the user of this methodology makes them someone else's problem, which can be wasteful of other teams' resources when it turns out not to be their problem after all. This anti-methodology can be identified by a lack of data leading to the hypothesis.

To avoid becoming a victim of blame-someone-else, ask the accuser for screenshots showing which tools were run and how the output was interpreted. You can take these screenshots and interpretations to someone else for a second opinion.

2.5.4 Ad Hoc Checklist Method

Stepping through a canned checklist is a common methodology used by support professionals when asked to check and tune a system, often in a short time frame. A typical scenario involves the deployment of a new server or application in production, and a support professional spending half a day checking for common issues now that the system is under real load. These checklists are ad hoc and are built from recent experience and issues for that system type.

Here is an example checklist entry:

> Run `iostat –x 1` and check the `r_await` column. If this is consistently over 10 (ms) during load, then either disk reads are slow or the disk is overloaded.

A checklist may be composed of a dozen or so such checks.

While these checklists can provide the most value in the shortest time frame, they are point-in-time recommendations (see Section 2.3, Concepts) and need to be frequently refreshed to stay current. They also tend to focus on issues for which there are known fixes that can be easily documented, such as the setting of tunable parameters, but not custom fixes to the source code or environment.

If you are managing a team of support professionals, an ad hoc checklist can be an effective way to ensure that everyone knows how to check for common issues. A checklist can be written to be clear and prescriptive, showing how to identify each issue and what the fix is. But bear in mind that this list must be constantly updated.

2.5.5 Problem Statement

Defining the problem statement is a routine task for support staff when first responding to issues. It's done by asking the customer the following questions:

1. What makes you think there is a performance problem?
2. Has this system ever performed well?
3. What changed recently? Software? Hardware? Load?
4. Can the problem be expressed in terms of latency or runtime?
5. Does the problem affect other people or applications (or is it just you)?
6. What is the environment? What software and hardware are used? Versions? Configuration?

Just asking and answering these questions often points to an immediate cause and solution. The problem statement has therefore been included here as its own methodology and should be the first approach you use when tackling a new issue.

I have solved performance issues over the phone by using the problem statement method alone, and without needing to log in to any server or look at any metrics.

2.5.6 Scientific Method

The scientific method studies the unknown by making hypotheses and then testing them. It can be summarized by the following steps:

1. Question
2. Hypothesis
3. Prediction
4. Test
5. Analysis

The question is the performance problem statement. From this you can hypothesize what the cause of poor performance may be. Then you construct a test, which may be observational or experimental, that tests a prediction based on the hypothesis. You finish with analysis of the test data collected.

For example, you may find that application performance is degraded after migrating to a system with less main memory, and you hypothesize that the cause of poor performance is a smaller file system cache. You might use an *observational test* to measure the cache miss rate on both

systems, predicting that cache misses will be higher on the smaller system. An *experimental test* would be to increase the cache size (adding RAM), predicting that performance will improve. Another, perhaps easier, experimental test is to artificially reduce the cache size (using tunable parameters), predicting that performance will be worse.

The following are some more examples.

Example (Observational)

1. Question: What is causing slow database queries?
2. Hypothesis: Noisy neighbors (other cloud computing tenants) are performing disk I/O, contending with database disk I/O (via the file system).
3. Prediction: If file system I/O latency is measured during a query, it will show that the file system is responsible for the slow queries.
4. Test: Tracing of database file system latency as a ratio of query latency shows that less than 5% of the time is spent waiting for the file system.
5. Analysis: The file system and disks are not responsible for slow queries.

Although the issue is still unsolved, some large components of the environment have been ruled out. The person conducting this investigation can return to step 2 and develop a new hypothesis.

Example (Experimental)

1. Question: Why do HTTP requests take longer from host A to host C than from host B to host C?
2. Hypothesis: Host A and host B are in different data centers.
3. Prediction: Moving host A to the same data center as host B will fix the problem.
4. Test: Move host A and measure performance.
5. Analysis: Performance has been fixed—consistent with the hypothesis.

If the problem wasn't fixed, reverse the experimental change (move host A back, in this case) before beginning a new hypothesis—changing multiple factors at once makes it harder to identify which one mattered!

Example (Experimental)

1. Question: Why did file system performance degrade as the file system cache grew in size?
2. Hypothesis: A larger cache stores more records, and more compute is required to manage a larger cache than a smaller one.
3. Prediction: Making the record size progressively smaller, and therefore causing more records to be used to store the same amount of data, will make performance progressively *worse*.
4. Test: Test the same workload with progressively smaller record sizes.

5. Analysis: Results are graphed and are consistent with the prediction. Drill-down analysis is now performed on the cache management routines.

This is an example of a *negative test*—deliberately hurting performance to learn more about the target system.

2.5.7 Diagnosis Cycle

Similar to the scientific method is the *diagnosis cycle*:

hypothesis → instrumentation → data → hypothesis

Like the scientific method, this method also deliberately tests a hypothesis through the collection of data. The cycle emphasizes that the data can lead quickly to a new hypothesis, which is tested and refined, and so on. This is similar to a doctor making a series of small tests to diagnose a patient and refining the hypothesis based on the result of each test.

Both of these approaches have a good balance of theory and data. Try to move from hypothesis to data quickly, so that bad theories can be identified early and discarded, and better ones developed.

2.5.8 Tools Method

A tools-oriented approach is as follows:

1. List available performance tools (optionally, install or purchase more).
2. For each tool, list useful metrics it provides.
3. For each metric, list possible ways to interpret it.

The result of this is a prescriptive checklist showing which tool to run, which metrics to read, and how to interpret them. While this can be fairly effective, it relies exclusively on available (or known) tools, which can provide an incomplete view of the system, similar to the streetlight anti-method. Worse, the user is unaware that they have an incomplete view—and may remain unaware. Issues that require custom tooling (e.g., dynamic tracing) may never be identified and solved.

In practice, the tools method does identify certain resource bottlenecks, errors, and other types of problems, though it may not do this efficiently.

When a large number of tools and metrics are available, it can be time-consuming to iterate through them. The situation gets worse when multiple tools appear to have the same functionality and you spend additional time trying to understand the pros and cons of each. In some cases, such as file system micro-benchmark tools, there are over a dozen tools to choose from, when you may need only one.[4]

[4] As an aside, an argument I've encountered to support multiple overlapping tools is that "competition is good." I would be cautious about this: while it can be helpful to have overlapping tools for cross-checking results (and I frequently cross-check BPF tools using Ftrace), multiple overlapping tools can become a waste of developer time that could be more effectively used elsewhere, as well as a waste of time for end users who must evaluate each choice.

2.5.9 The USE Method

The utilization, saturation, and errors (USE) method should be used early in a performance investigation to identify systemic bottlenecks [Gregg 13b]. It is a methodology that focuses on system resources and can be summarized as:

> For every resource, check utilization, saturation, and errors.

These terms are defined as follows:

- **Resources**: All physical server functional components (CPUs, buses, . . .). Some software resources can also be examined, provided that the metrics make sense.
- **Utilization**: For a set time interval, the percentage of time that the resource was busy servicing work. While busy, the resource may still be able to accept more work; the degree to which it cannot do so is identified by saturation.
- **Saturation**: The degree to which the resource has extra work that it can't service, often waiting on a queue. Another term for this is *pressure*.
- **Errors**: The count of error events.

For some resource types, including main memory, utilization is the *capacity* of the resource that is used. This is different from the time-based definition and was explained earlier in Section 2.3.11, Utilization. Once a capacity resource reaches 100% utilization, more work cannot be accepted, and the resource either queues the work (saturation) or returns errors, which are also identified using the USE method.

Errors should be investigated because they can degrade performance but may not be immediately noticed when the failure mode is recoverable. This includes operations that fail and are retried, and devices that fail in a pool of redundant devices.

In contrast with the tools method, the USE method involves iterating over system resources instead of tools. This helps you create a complete list of questions to ask, and only then do you search for tools to answer them. Even when tools cannot be found to answer some questions, the knowledge that these questions are unanswered can be extremely useful for the performance analyst: they are now "known-unknowns."

The USE method also directs analysis to a limited number of key metrics, so that all system resources are checked as quickly as possible. After this, if no issues have been found, other methodologies can be used.

Procedure

The USE method is pictured as the flowchart in Figure 2.12. Errors are checked first because they are usually quick to interpret (they are usually an objective and not subjective metric), and it can be time-efficient to rule them out before investigating the other metrics. Saturation is checked second because it is quicker to interpret than utilization: any level of saturation can be an issue.

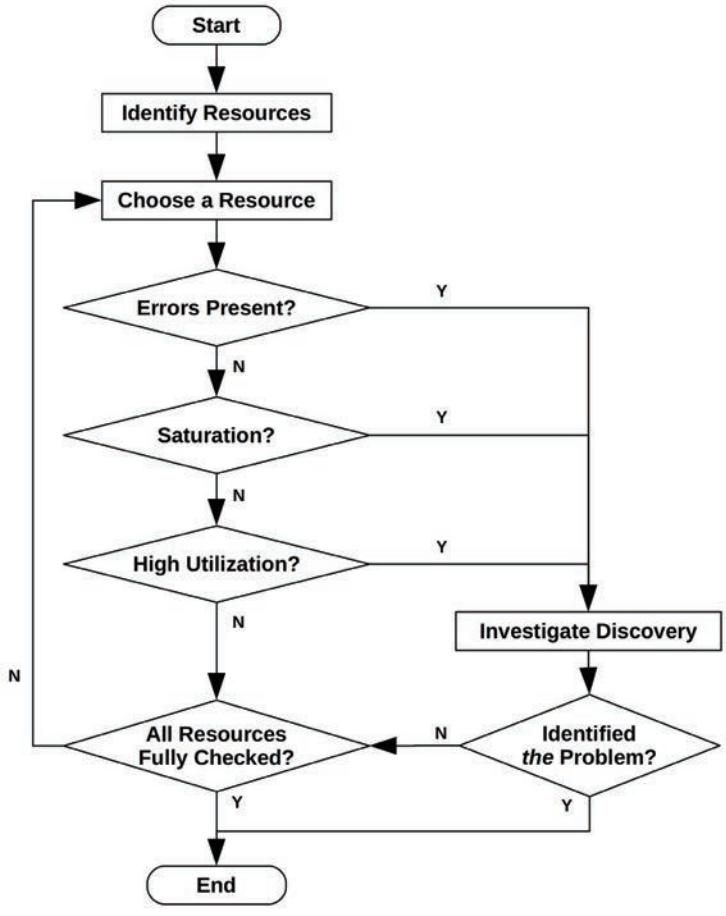


Figure 2.12 The USE method flow

This method identifies problems that are likely to be system bottlenecks. Unfortunately, a system may be suffering from more than one performance problem, so the first thing you find may be *a* problem but not *the* problem. Each discovery can be investigated using further methodologies, before returning to the USE method as needed to iterate over more resources.

Expressing Metrics

The USE method metrics are usually expressed as follows:

- **Utilization**: As a percent over a time interval (e.g., "One CPU is running at 90% utilization")
- **Saturation**: As a wait-queue length (e.g., "The CPUs have an average run-queue length of four")
- **Errors**: Number of errors reported (e.g., "This disk drive has had 50 errors")

Though it may seem counterintuitive, a short burst of high utilization can cause saturation and performance issues, even though the overall utilization is *low* over a long interval. Some monitoring tools report utilization over 5-minute averages. CPU utilization, for example, can vary dramatically from second to second, so a 5-minute average may disguise short periods of 100% utilization and, therefore, saturation.

Consider a toll plaza on a highway. Utilization can be defined as how many tollbooths were busy servicing a car. Utilization at 100% means you can't find an empty booth and must queue behind someone (saturation). If I told you the booths were at 40% utilization across the entire day, could you tell me whether any cars had queued at any time during that day? They probably did during rush hour, when utilization was at 100%, but that isn't visible in the daily average.

Resource List

The first step in the USE method is to create a list of resources. Try to be as complete as possible. Here is a generic list of server hardware resources, along with specific examples:

- **CPUs**: Sockets, cores, hardware threads (virtual CPUs)
- **Main memory**: DRAM
- **Network interfaces**: Ethernet ports, Infiniband
- **Storage devices**: Disks, storage adapters
- **Accelerators**: GPUs, TPUs, FPGAs, etc., if in use
- **Controllers**: Storage, network
- **Interconnects**: CPU, memory, I/O

Each component typically acts as a single resource type. For example, main memory is a *capacity* resource, and network interfaces are an *I/O* resource (which can mean either IOPS or throughput). Some components can behave as multiple resource types: for example, a storage device is both an I/O resource and a capacity resource. Consider all types that can lead to performance bottlenecks. Also note that I/O resources can be further studied as *queueing systems*, which queue and then service these requests.

Some physical components, such as hardware caches (e.g., CPU caches), can be left out of your checklist. The USE method is most effective for resources that suffer performance degradation under high utilization or saturation, leading to bottlenecks, while caches *improve* performance under high utilization. These can be checked using other methodologies. If you are unsure whether to include a resource, include it, and then see how well the metrics work in practice.

Functional Block Diagram

Another way to iterate over resources is to find or draw a functional block diagram for the system, such as the one shown in Figure 2.13. Such a diagram also shows relationships, which can be very useful when looking for bottlenecks in the flow of data.

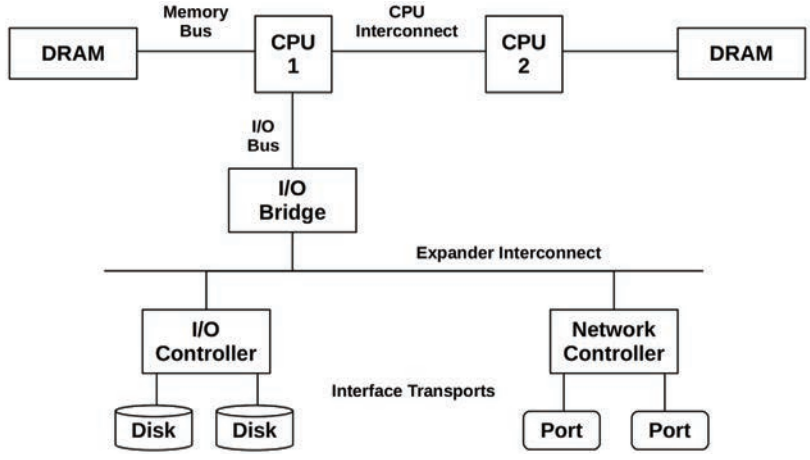


Figure 2.13 Example two-processor functional block diagram

CPU, memory, and I/O interconnects and buses are often overlooked. Fortunately, they are not common system bottlenecks, as they are typically designed to provide an excess of throughput. Unfortunately, if they are, the problem can be difficult to solve. Maybe you can upgrade the main board, or reduce load; for example, "zero copy" software techniques lighten memory bus load.

To investigate interconnects, see CPU Performance Counters in Chapter 6, CPUs, Section 6.4.1, Hardware.

Metrics

Once you have your list of resources, consider the metric types appropriate to each: utilization, saturation, and errors. Table 2.6 shows some example resources and metric types, along with possible metrics (generic OS).

Table 2.6 **Example USE method metrics**

Resource	Type	Metric
CPU	Utilization	CPU utilization (either per CPU or a system-wide average)
CPU	Saturation	Run queue length, scheduler latency, CPU pressure (Linux PSI)
Memory	Utilization	Available free memory (system-wide)
Memory	Saturation	Swapping (anonymous paging), page scanning, out-of-memory events, memory pressure (Linux PSI)
Network interface	Utilization	Receive throughput/max bandwidth, transmit throughput/max bandwidth
Storage device I/O	Utilization	Device busy percent

Resource	Type	Metric
Storage device I/O	Saturation	Wait queue length, I/O pressure (Linux PSI)
Storage device I/O	Errors	Device errors (“soft,” “hard”)

These metrics can be either averages per interval or counts.

Repeat for all combinations, and include instructions for fetching each metric. Take note of metrics that are not currently available; these are the known-unknowns. You’ll end up with a list of about 30 metrics, some of which are difficult to measure, and some of which can’t be measured at all. Fortunately, the most common issues are usually found with the easier metrics (e.g., CPU saturation, memory capacity saturation, network interface utilization, disk utilization), so these can be checked first.

Some examples of harder combinations are provided in Table 2.7.

Table 2.7 **Example USE method advanced metrics**

Resource	Type	Metric
CPU	Errors	For example, machine check exceptions, CPU cache errors[5]
Memory	Errors	For example, failed malloc()s (although a default Linux kernel configuration makes this rare due to overcommit)
Network	Saturation	Saturation-related network interface or OS errors, e.g., Linux “overruns”
Storage controller	Utilization	Depends on the controller; it may have a maximum IOPS or throughput that can be checked against current activity
CPU interconnect	Utilization	Per-port throughput/maximum bandwidth (CPU performance counters)
Memory interconnect	Saturation	Memory stall cycles, high cycles per instruction (CPU performance counters)
I/O interconnect	Utilization	Bus throughput/maximum bandwidth (performance counters may exist on your HW, e.g., Intel “uncore” events)

Some of these may not be available from standard operating system tools and may require the use of dynamic tracing or CPU performance monitoring counters.

Appendix A is an example USE method checklist for Linux systems, iterating over all combinations for hardware resources with the Linux observability toolset, and includes some software resources, such as those described in the next section.

[5] For example, recoverable error-correcting code (ECC) errors for CPU cache lines (if supported). Some kernels will offline a CPU if an increase in these is detected.

Software Resources

Some software resources can be similarly examined. This usually applies to smaller components of software (not entire applications), for example:

- **Mutex locks**: Utilization may be defined as the time the lock was held, saturation by those threads queued waiting on the lock.
- **Thread pools**: Utilization may be defined as the time threads were busy processing work, saturation by the number of requests waiting to be serviced by the thread pool.
- **Process/thread capacity**: The system may have a limited number of processes or threads, whose current usage may be defined as utilization; waiting on allocation may be saturation; and errors are when the allocation failed (e.g., "cannot fork").
- **File descriptor capacity**: Similar to process/thread capacity, but for file descriptors.

If the metrics work well in your case, use them; otherwise, alternative methodologies such as latency analysis can be applied.

Suggested Interpretations

Here are some general suggestions for interpreting the metric types:

- **Utilization**: Utilization at 100% is usually a sign of a bottleneck (check saturation and its effect to confirm). Utilization beyond 60% can be a problem for a couple of reasons: depending on the interval, it can hide short bursts of 100% utilization. Also, some resources such as hard disks (but not CPUs) usually cannot be interrupted during an operation, even for higher-priority work. As utilization increases, queueing delays become more frequent and noticeable. See Section 2.6.5, Queueing Theory, for more about 60% utilization.
- **Saturation**: Any degree of saturation (non-zero) can be a problem. It may be measured as the length of a wait queue, or as time spent waiting on the queue.
- **Errors**: Non-zero error counters are worth investigating, especially if they are increasing while performance is poor.

It's easy to interpret the negative cases: low utilization, no saturation, no errors. This is more useful than it sounds—narrowing down the scope of an investigation can help you focus quickly on the problem area, having identified that it is likely *not* a resource problem. This is the process of elimination.

Resource Controls

In cloud computing and container environments, software resource controls may be in place to limit or throttle tenants who are sharing one system. These may limit memory, CPU, disk I/O, and network I/O. For example, Linux containers use cgroups to limit resource usage. Each of these resource limits can be examined with the USE method, similarly to examining the physical resources.

For example, "memory capacity utilization" can be the tenant's memory usage versus its memory cap. "Memory capacity saturation" can be seen by limit-imposed allocation errors or swapping for that tenant, even if the host system is not experiencing memory pressure. These limits are discussed in Chapter 11, Cloud Computing.

Microservices

A microservice architecture presents a similar problem to that of too many resource metrics: there can be so many metrics for each service that it is laborious to check them all, and they can overlook areas where metrics do not yet exist. The USE method can address these problems with microservices as well. For example, for a typical Netflix microservice, the USE metrics are:

- **Utilization**: The average CPU utilization across the entire instance cluster.
- **Saturation**: An approximation is the difference between the 99th latency percentile and the average latency (assumes the 99th is saturation-driven).
- **Errors**: Request errors.

These three metrics are already examined for each microservice at Netflix using the Atlas cloud-wide monitoring tool [Harrington 14].

There is a similar methodology that has been designed specifically for services: the RED method.

2.5.10 The RED Method

The focus of this methodology is services, typically cloud services in a microservice architecture. It identifies three metrics for monitoring health from a user perspective and can be summarized as [Wilkie 18]:

> For every service, check the request rate, errors, and duration.

The metrics are:

- **Request rate**: The number of service requests per second
- **Errors**: The number of requests that failed
- **Duration**: The time for requests to complete (consider distribution statistics such as percentiles in addition to the average: see Section 2.8, Statistics)

Your task is to draw a diagram of your microservice architecture and ensure that these three metrics are monitored for each service. (Distributed tracing tools may provide such diagrams for you.) The advantages are similar to the USE method: the RED method is fast and easy to follow, and comprehensive.

The RED method was created by Tom Wilkie, who has also developed implementations of the USE and RED method metrics for Prometheus with dashboards using Grafana [Wilkie 18]. These methodologies are complementary: the USE method for machine health, and the RED method for user health.

The inclusion of the request rate provides an important early clue in an investigation: whether a performance problem is one of load versus architecture (see Section 2.3.8, Load vs. Architecture). If the request rate has been steady but the request duration has increased, it points to a problem with the architecture: the service itself. If both the request rate and duration have increased, then the problem may be one of the load applied. This can be further investigated using workload characterization.

2.5.11 Workload Characterization

Workload characterization is a simple and effective method for identifying a class of issues: those due to the load applied. It focuses on the *input* to the system, rather than the resulting performance. Your system may have no architectural, implementation, or configuration issues present, but be experiencing more load than it can reasonably handle.

Workloads can be characterized by answering the following questions:

- **Who** is causing the load? Process ID, user ID, remote IP address?
- **Why** is the load being called? Code path, stack trace?
- **What** are the load characteristics? IOPS, throughput, direction (read/write), type? Include variance (standard deviation) where appropriate.
- **How** is the load changing over time? Is there a daily pattern?

It can be useful to check all of these, even when you have strong expectations about what the answers will be, because you may be surprised.

Consider this scenario: You have a performance issue with a database whose clients are a pool of web servers. Should you check the IP addresses of who is using the database? You already expect them to be the web servers, as per the configuration. You check anyway and discover that the entire Internet appears to be throwing load at the databases, destroying their performance. You are actually under a denial-of-service (DoS) attack!

The best performance wins are the result of *eliminating unnecessary work*. Sometimes unnecessary work is caused by applications malfunctioning, for example, a thread stuck in a loop creating unnecessary CPU work. It can also be caused by bad configurations—for example, system-wide backups that run during peak hours—or even a DoS attack as described previously. Characterizing the workload can identify these issues, and with maintenance or reconfiguration they may be eliminated.

If the identified workload cannot be eliminated, another approach may be to use system resource controls to throttle it. For example, a system backup task may be interfering with a production database by consuming CPU resources to compress the backup, and then network resources to transfer it. This CPU and network usage may be throttled using resource controls (if the system supports them) so that the backup runs more slowly without hurting the database.

Apart from identifying issues, workload characterization can also be input for the design of simulation benchmarks. If the workload measurement is an average, ideally you will also collect details of the distribution and variation. This can be important for simulating the variety of workloads expected, rather than testing only an average workload. See Section 2.8, Statistics, for more about averages and variation (standard deviation), and Chapter 12, Benchmarking.

Analysis of the workload also helps separate problems of load from problems of architecture, by identifying the former. Load versus architecture was introduced in Section 2.3.8, Load vs. Architecture.

The specific tools and metrics for performing workload characterization depend on the target. Some applications record detailed logs of client activity, which can be the source for statistical analysis. They may also already provide daily or monthly reports of client usage, which can be mined for details.