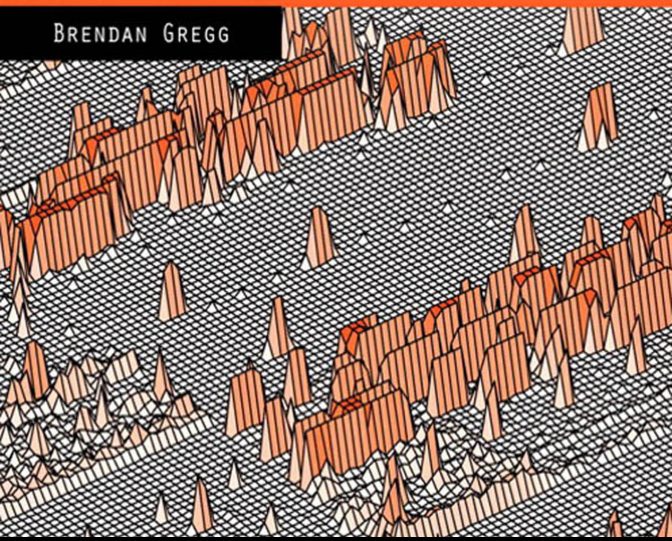
PRENTICE HALL

Systems Performance

Enterprise and the Cloud

Brendan Gregg



FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Systems Performance

This page intentionally left blank

Systems Performance

Enterprise and the Cloud

Brendan Gregg

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data

Gregg, Brendan.
Systems performance : enterprise and the cloud / Brendan Gregg.
pages cm
Includes bibliographical references and index.
ISBN-13: 978-0-13-339009-4 (alkaline paper)
ISBN-10: 0-13-339009-8 (alkaline paper)
1. Operating systems (Computers)—Evaluation. 2. Application software—Evaluation. 3. Business Enterprises—Data processing. 4. Cloud computing. I. Title.
QA76.77.G74 2014
004.67'82—dc23

2013031887

Copyright © 2014 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-13-339009-4
ISBN-10: 0-13-339009-8
Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
Second Printing, January 2014

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 above 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 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 Book

Welcome to *Systems Performance: Enterprise and the Cloud*! This book is about the performance of operating systems and of applications from operating system context, and it is written for both enterprise and cloud computing environments. My aim is to help you get the most out of your systems.

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. 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 in the operating system context; I'll cover that here as well.

Operating System Coverage

The main focus of this book is the study of systems performance, with tools, examples, and tunable parameters from Linux- and Solaris-based operating systems used as examples. Unless noted, the specific distribution of an operating system is not important in the examples used. For Linux-based systems, the examples are from a variety of bare-metal systems and virtualized cloud tenants, running either Ubuntu, Fedora, or CentOS. For Solaris-based systems, the examples are also either bare-metal or virtualized and are from either Joyent SmartOS or OmniTI OmniOS. SmartOS and OmniOS use the open-source illumos kernel: the active fork of the OpenSolaris kernel, which itself was based on the development version of what became Oracle Solaris 11.

Covering two different operating systems provides an additional perspective for each audience, offering a deeper understanding of their characteristics, especially where each OS has taken a different design path. This helps the reader to understand performance more comprehensively, without being limited to a single OS, and to think about operating systems more objectively.

Historically, more performance work has been done for Solaris-based systems, making them the better choice for some examples. The situation for Linux has been greatly improving. When *System Performance Tuning* [Musumeci 02] was written, over a decade ago, it also addressed both Linux and Solaris but was heavily oriented toward the latter. The author noted reasons for this:

> Solaris machines tend to be more focused on performance. I suspect this is because Sun systems are more expensive than their Linux counterparts, on average. As a result, people tend to be a lot more picky about performance, so more work has been done in that area on Solaris. If your Linux box doesn't perform well enough, you can just buy another one and split up the workload—it's cheap. If your several-million-dollar Ultra Enterprise 10000 doesn't perform well and your company is losing non-trivial sums of money every minute because of it, you call Sun Service and start demanding answers.

This helps explain Sun's historical performance focus: Solaris profits were tied to hardware sales, and real money was frequently on the line for performance improvements. Sun needed—and could afford to hire—over 100 full-time performance engineers (including, at times, myself and Musumeci). Together with Sun's kernel engineering teams, many developments were made in the field of systems performance.

Linux has come a long way in terms of performance work and observability, especially now that it is being used in large-scale cloud computing environments. Many performance features for Linux, included in this book, have been developed only within the past five years.

Other Content

Example screen shots 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 ways, many in the style of earlier Unix tools, producing output that is familiar and often self-explanatory. This means that screen shots 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!)

The history of technologies can provide useful insight to deepen your understanding, and it has been mentioned in places. It is also useful to learn a bit about the key people in this industry (it's a small world): 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 G.

What Isn't Covered

This book focuses on performance. To perform 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). Specifically on Linux, you will need to install the sysstat package, as many of its tools are used in this text.

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 then study them from additional sources if and when needed.

How This Book Is Structured

The book includes the following:

- **Chapter 1, Introduction,** is an introduction to systems performance analysis, summarizing key concepts and providing examples of performance activities.
- **Chapter 2, Methodology,** 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, and kernel scheduling.
- **Chapter 7, Memory,** is about virtual memory, paging, swapping, memory architectures, busses, 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 and their performance overhead, isolation, and observability characteristics.
- **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, Case Study,** contains a systems performance case study, showing how a real cloud customer issue 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.

Chapter 13 is written differently, using 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, for context, and then return to it again 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 in the context of examples.

Tracing Examples

We frequently need to explore the operating system in depth, which can be performed by kernel tracing tools. There are many of these at various stages of development, for example, ftrace, perf, DTrace, SystemTap, LTTng, and ktap. One of them has been chosen for most of the tracing examples here and is demonstrated on both Linux- and Solaris-based systems: DTrace. It provides the features needed for these examples, and there is also a large amount of external material about it, including scripts that can be referenced as use cases of advanced tracing.

You may need or wish to use different tracing tools, which is fine. The DTrace examples are examples of tracing and 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 the lead performance engineer at a cloud computing provider, I frequently work with support staff and customers who are under enormous time pressure to solve multiple performance issues. For many, performance is not their primary job, and they need to know just enough to solve the issues at hand. This has encouraged me to keep this book as short as possible, knowing that your time to study it may be very limited. But not too short: there is much to cover to ensure that you are prepared.

Another intended audience is students: this book is also suitable as a supporting text for a systems performance course. During the writing of this book (and for many years before it began), I developed and taught such classes myself, which included simulated performance issues for the students to solve (without providing the answers beforehand!). This has helped me to see which types of material work best in leading students to solve performance problems, and that has guided my choice of content for this book.

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

In terms of company size, this book should contain enough detail to satisfy small to large environments, 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:

`netif_receive_skb()`	function name
`iostat(1)`	man page
Documentation/ . . .	Linux docs
CONFIG_ . . .	Linux configuration option
kernel/ . . .	Linux kernel source code
fs/	Linux kernel source code, file systems
usr/src/uts/ . . .	Solaris-based kernel source code
#	superuser (root) shell prompt
$	user (non-root) shell prompt
^C	a command was interrupted (Ctrl-C)
`[...]`	truncation
`mpstat 1`	typed command or highlighting

Supplemental Material and References

The following selected texts (the full list is in the Bibliography) can 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 R. Pettit. *Sun Performance and Tuning: Java and the Internet*. Prentice Hall, 1998.

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

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

[McDougall 06a] McDougall, R., and J. Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Prentice Hall, 2006.

[McDougall 06b] McDougall, R., J. Mauro, and B. Gregg. *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 11] Gregg, B., and J. Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall, 2011.

This page intentionally left blank

Acknowledgments

Deirdré Straughan has, once again, provided amazing help, sharing my interest in technical education deeply enough to survive another book. She has been involved from concept to manuscript, at first helping me plan what this book would be, then spending countless hours editing and discussing every draft page, identifying many parts I hadn't explained properly. At this point I've worked with her on over 2,000 pages of technical content (plus blog posts!), and I'm lucky to have had such outstanding help.

Barbara Wood performed the copy edit and worked through the text in great detail and in great time, making numerous final improvements to its quality, readability, and consistency. With the length and complexity, this is a difficult text to work on, and I'm very glad for Barbara's help and hard work.

I'm very grateful for everyone who provided feedback on some or all of the book. This is a deeply technical book with many new topics and has required serious effort to review the material—frequently requiring kernel source code from different kernels to be double-checked and understood.

Darryl Gove provided outstanding feedback, both at a deeply technical level and for the high-level presentation and organization of material. He is an author himself, and I look forward to any of his future books, knowing how driven he is to provide the best possible material to our readers.

I'm very grateful to Richard Lowe and Robert Mustacchi, who both worked through the entire book and found topics I had missed or needed to explain better. Richard's understanding of different kernel internals is astonishing, and also a

little terrifying. Robert also helped considerably with the Cloud Computing chapter, bringing to bear his expertise from working on the KVM port to illumos.

Thanks for the feedback from Jim Mauro and Dominic Kay: I've worked with them on books before, and they have great minds for comprehending difficult technical content and then explaining it to readers.

Jerry Jelinek and Max Bruning, both of whom have kernel engineering expertise, also provided detailed feedback on multiple chapters.

Adam Leventhal provided expert feedback for the File Systems and Disks chapters, notably helping me to understand the current nuances of flash memory—an area where he has longstanding expertise, having invented innovative new uses of flash memory while at Sun.

David Pacheco provided excellent feedback on the Applications chapter, and Dan McDonald on the Network chapter. I'm lucky to have them bring their expertise to areas they know so well.

Carlos Cardenas worked through the entire book and provided some unique feedback that I was seeking regarding statistical analysis.

I'm grateful to Bryan Cantrill, Keith Wesolowski, Paul Eggleton, Marsell Kukuljevic-Pearce, and Adrian Cockcroft, for their feedback and contributions. Adrian's comments encouraged me to reshuffle the chapter order, helping the reader better relate to the material covered.

I'm grateful to authors before me, whose names are listed in the Bibliography, who have forged paths into systems performance and documented their findings. I've also captured expertise I've learned from performance experts I've worked with over the years, including Bryan Cantrill, Roch Bourbonnais, Jim Mauro, Richard McDougall, and many others, from whom I've learned much.

Thanks to Bryan Cantrill for supporting this project, and to Jason Hoffman for his enthusiasm.

Thanks to Claire, Mitchell, and other family and friends for making the sacrifices to support me in a project like this.

And a special thanks to Greg Doench, senior editor at Pearson, for his help, patience, and advice on the project.

I've enjoyed working on this book, though it has at times been daunting. It would have been much easier for me to write it over a decade ago, when I knew less about the complexities and subtle nuances of systems performance. Since then, I've worked as a software engineer, a kernel engineer, and a performance engineer, and in enterprise, storage, and cloud computing. I've debugged performance issues everywhere in the stack, from applications to metal. This experience, and knowing how much has not yet been documented, has both discouraged and encouraged me to write about it. This is the book I thought needed to be written, and it's a relief to have it done.

About the Author

Brendan Gregg is the lead performance engineer at Joyent, where he analyzes performance and scalability for small to large cloud computing environments, at any level of the software stack. He is the primary author of *DTrace* (Prentice Hall, 2011), and coauthor of *Solaris Performance and Tools* (Prentice Hall, 2007), as well as numerous articles about systems performance. He was previously a performance lead and kernel engineer at Sun Microsystems, and also a performance consultant and trainer. He developed the DTraceToolkit and the ZFS L2ARC, and many of his DTrace scripts are shipped by default in Mac OS X and Oracle Solaris 11. His recent work has included performance visualizations.

This page intentionally left blank

This page intentionally left blank

6

CPUs

CPUs drive all software and are often the first target for systems performance analysis. Modern systems typically have many CPUs, which are shared among all running software by the kernel scheduler. When there is more demand for CPU resources than there are resources available, process threads (or tasks) will queue, waiting their turn. Waiting can add significant latency during the runtime of applications, degrading performance.

The usage of the CPUs can be examined in detail to look for performance improvements, including eliminating unnecessary work. At a high level, CPU usage by process, thread, or task can be examined. At a lower level, the code path within applications and the kernel can be profiled and studied. At the lowest level, CPU instruction execution and cycle behavior can be studied.

This chapter consists of five parts:

- **Background** introduces CPU-related terminology, basic models of CPUs, and key CPU performance concepts.
- **Architecture** introduces processor and kernel scheduler architecture.
- **Methodology** describes performance analysis methodologies, both observational and experimental.
- **Analysis** describes CPU performance analysis tools on Linux- and Solaris-based systems, including profiling, tracing, and visualizations.
- **Tuning** includes examples of tunable parameters.

The first three sections provide the basis for CPU analysis, and the last two show its practical application to Linux- and Solaris-based systems.

The effects of memory I/O on CPU performance are covered, including CPU cycles stalled on memory and the performance of CPU caches. Chapter 7, Memory, continues the discussion of memory I/O, including MMU, NUMA/UMA, system interconnects, and memory busses.

6.1 Terminology

For reference, CPU-related terminology used in this chapter includes the following:

- **Processor:** the physical chip that plugs into a socket on the system or processor board and contains one or more CPUs implemented as cores or hardware threads.
- **Core:** an independent CPU instance on a *multicore processor*. The use of cores is a way to scale processors, called *chip-level multiprocessing* (CMP).
- **Hardware thread:** a CPU architecture that supports executing multiple threads in parallel on a single core (including Intel's Hyper-Threading Technology), where each thread is an independent CPU instance. One name for this scaling approach is *multithreading*.
- **CPU instruction:** a single CPU operation, from its *instruction set*. There are instructions for arithmetic operations, memory I/O, and control logic.
- **Logical CPU:** also called a *virtual processor*,[1] an operating system CPU instance (a schedulable CPU entity). This may be implemented by the processor as a hardware thread (in which case it may also be called a *virtual core*), a core, or a single-core processor.
- **Scheduler:** the kernel subsystem that assigns threads to run on CPUs.
- **Run queue:** a queue of runnable threads that are waiting to be serviced by CPUs. For Solaris, it is often called a *dispatcher queue*.

Other terms are introduced throughout this chapter. The Glossary includes basic terminology for reference, including *CPU*, *CPU cycle*, and *stack*. Also see the terminology sections in Chapters 2 and 3.

1. It is also sometimes called a *virtual CPU*; however, that term is more commonly used to refer to virtual CPU instances provided by a virtualization technology. See Chapter 11, Cloud Computing.

6.2 Models

The following simple models illustrate some basic principles of CPUs and CPU performance. Section 6.4, Architecture, digs much deeper and includes implementation-specific details.

6.2.1 CPU Architecture

Figure 6.1 shows an example CPU architecture, for a single processor with four cores and eight hardware threads in total. The physical architecture is pictured, along with how it is seen by the operating system.

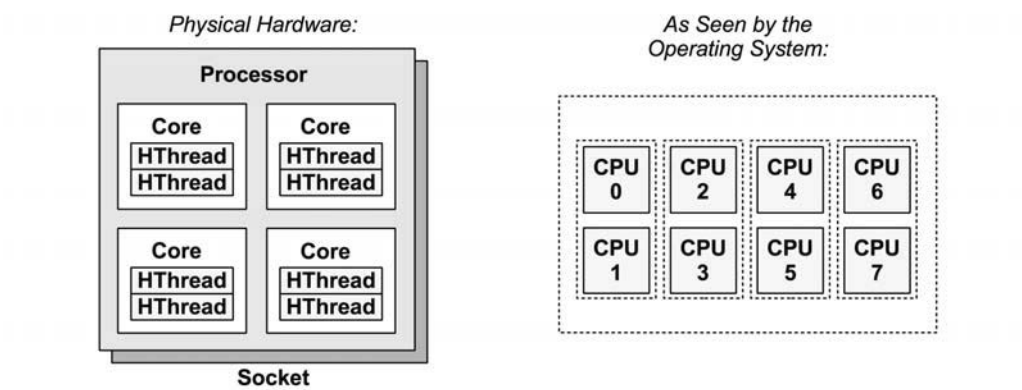


Figure 6-1 CPU architecture

Each hardware thread is addressable as a *logical CPU*, so this processor appears as eight CPUs. The operating system may have some additional knowledge of topology, such as which CPUs are on the same core, to improve its scheduling decisions.

6.2.2 CPU Memory Caches

Processors provide various hardware caches for improving memory I/O performance. Figure 6.2 shows the relationship of cache sizes, which become smaller and faster (a trade-off) the closer they are to the CPU.

The caches that are present, and whether they are on the processor (integrated) or external to the processor, depend on the processor type. Earlier processors provided fewer levels of integrated cache.

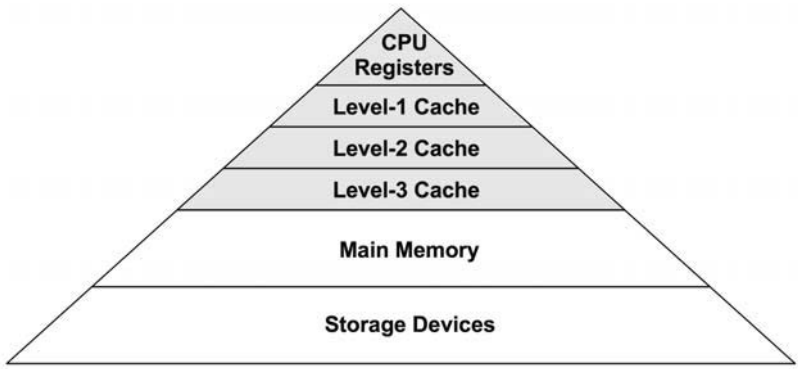


Figure 6-2 CPU cache sizes

6.2.3 CPU Run Queues

Figure 6.3 shows a CPU run queue, which is managed by the kernel scheduler.

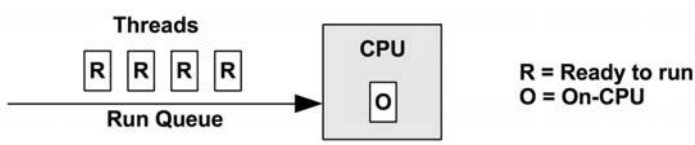


Figure 6-3 CPU run queue

The thread states shown in the figure, ready to run and on-CPU, are covered in Figure 3.7 in Chapter 3, Operating Systems.

The number of software threads that are queued and ready to run is an important performance metric indicating CPU saturation. In this figure (at this instant) there are four, with an additional thread running on-CPU. The time spent waiting on a CPU run queue is sometimes called *run-queue latency* or *dispatcher-queue latency*. In this book, the term *scheduler latency* is used instead, as it is appropriate for all dispatcher types, including those that do not use queues (see the discussion of CFS in Section 6.4.2, Software).

For multiprocessor systems, the kernel typically provides a run queue for each CPU and aims to keep threads on the same run queue. This means that threads are more likely to keep running on the same CPUs, where the CPU caches have cached their data. (These caches are described as having *cache warmth*, and the approach to favor CPUs is called *CPU affinity*.) On NUMA systems, *memory locality* may also be improved, which also improves performance (this is described in Chapter 7, Memory).

It also avoids the cost of thread synchronization (mutex locks) for queue operations, which would hurt scalability if the run queue was global and shared among all CPUs.

6.3 Concepts

The following are a selection of important concepts regarding CPU performance, beginning with a summary of processor internals: the CPU clock rate and how instructions are executed. This is background for later performance analysis, particularly for understanding the cycles-per-instruction (CPI) metric.

6.3.1 Clock Rate

The clock is a digital signal that drives all processor logic. Each CPU instruction may take one or more cycles of the clock (called *CPU cycles*) to execute. CPUs execute at a particular clock rate; for example, a 5 GHz CPU performs 5 billion clock cycles per second.

Some processors are able to vary their clock rate, increasing it to improve performance or decreasing it to reduce power consumption. The rate may be varied on request by the operating system, or dynamically by the processor itself. The kernel idle thread, for example, can request the CPU to throttle down to save power.

Clock rate is often marketed as the primary feature of the processor, but this can be a little misleading. Even if the CPU in your system appears to be fully utilized (a bottleneck), a faster clock rate may not speed up performance—it depends on what those fast CPU cycles are actually doing. If they are mostly stall cycles while waiting on memory access, executing them more quickly doesn't actually increase the CPU instruction rate or workload throughput.

6.3.2 Instruction

CPUs execute instructions chosen from their instruction set. An instruction includes the following steps, each processed by a component of the CPU called a *functional unit*:

1. Instruction fetch
2. Instruction decode
3. Execute
4. Memory access
5. Register write-back

The last two steps are optional, depending on the instruction. Many instructions operate on registers only and do not require the memory access step.

Each of these steps takes at least a single clock cycle to be executed. Memory access is often the slowest, as it may take dozens of clock cycles to read or write to main memory, during which instruction execution has *stalled* (and these cycles while stalled are called *stall cycles*). This is why CPU caching is important, as described in Section 6.4: it can dramatically reduce the number of cycles needed for memory access.

6.3.3 Instruction Pipeline

The instruction pipeline is a CPU architecture that can execute multiple instructions in parallel, by executing different components of different instructions at the same time. It is similar to a factory assembly line, where stages of production can be executed in parallel, increasing throughput.

Consider the instruction steps previously listed. If each were to take a single clock cycle, it would take five cycles to complete the instruction. At each step of this instruction, only one functional unit is active and four are idle. By use of pipelining, multiple functional units can be active at the same time, processing different instructions in the pipeline. Ideally, the processor can then complete one instruction with every clock cycle.

6.3.4 Instruction Width

But we can go faster still. Multiple functional units can be included of the same type, so that even more instructions can make forward progress with each clock cycle. This CPU architecture is called *superscalar* and is typically used with pipelining to achieve a high instruction throughput.

The instruction *width* describes the target number of instructions to process in parallel. Modern processors are *3-wide* or *4-wide*, meaning they can complete up to three or four instructions per cycle. How this works depends on the processor, as there may be different numbers of functional units for each stage.

6.3.5 CPI, IPC

Cycles per instruction (CPI) is an important high-level metric for describing where a CPU is spending its clock cycles and for understanding the nature of CPU utilization. This metric may also be expressed as *instructions per cycle* (IPC), the inverse of CPI.

A high CPI indicates that CPUs are often stalled, typically for memory access. A low CPI indicates that CPUs are often not stalled and have a high instruction throughput. These metrics suggest where performance tuning efforts may be best spent.

Memory-intensive workloads, for example, may be improved by installing faster memory (DRAM), improving memory locality (software configuration), or reducing the amount of memory I/O. Installing CPUs with a higher clock rate may not improve performance to the degree expected, as the CPUs may need to wait the same amount of time for memory I/O to complete. Put differently, a faster CPU may mean more stall cycles but the same rate of completed instructions.

The actual values for high or low CPI are dependent on the processor and processor features and can be determined experimentally by running known workloads. As an example, you may find that high-CPI workloads run with a CPI at ten or higher, and low CPI workloads run with a CPI at less than one (which is possible due to instruction pipelining and width, described earlier).

It should be noted that CPI shows the efficiency of instruction *processing*, but not of the instructions themselves. Consider a software change that added an inefficient software loop, which operates mostly on CPU registers (no stall cycles): such a change may result in a lower overall CPI, but higher CPU usage and utilization.

6.3.6 Utilization

CPU utilization is measured by the time a CPU instance is busy performing work during an interval, expressed as a percentage. It can be measured as the time a CPU is not running the kernel idle thread but is instead running user-level application threads or other kernel threads, or processing interrupts.

High CPU utilization may not necessarily be a problem, but rather a sign that the system is doing work. Some people also consider this an ROI indicator: a highly utilized system is considered to have good ROI, whereas an idle system is considered wasted. Unlike with other resource types (disks), performance does not degrade steeply under high utilization, as the kernel supports priorities, preemption, and time sharing. These together allow the kernel to understand what has higher priority, and to ensure that it runs first.

The measure of CPU utilization spans all clock cycles for eligible activities, including memory stall cycles. It may seem a little counterintuitive, but a CPU may be highly utilized because it is often stalled waiting for memory I/O, not just executing instructions, as described in the previous section.

CPU utilization is often split into separate kernel- and user-time metrics.

6.3.7 User-Time/Kernel-Time

The CPU time spent executing user-level application code is called *user-time*, and kernel-level code is *kernel-time*. Kernel-time includes time during system calls, kernel threads, and interrupts. When measured across the entire system, the user-time/kernel-time ratio indicates the type of workload performed.

Applications that are computation-intensive may spend almost all their time executing user-level code and have a user/kernel ratio approaching 99/1. Examples include image processing, genomics, and data analysis.

Applications that are I/O-intensive have a high rate of system calls, which execute kernel code to perform the I/O. For example, a web server performing network I/O may have a user/kernel ratio of around 70/30.

These numbers are dependent on many factors and are included to express the kinds of ratios expected.

6.3.8 Saturation

A CPU at 100% utilization is *saturated*, and threads will encounter *scheduler latency* as they wait to run on-CPU, decreasing overall performance. This latency is the time spent waiting on the CPU run queue or other structure used to manage threads.

Another form of CPU saturation involves CPU resource controls, as may be imposed in a multitenant cloud computing environment. While the CPU may not be 100% utilized, the imposed limit has been reached, and threads that are runnable must wait their turn. How visible this is to users of the system depends on the type of virtualization in use; see Chapter 11, Cloud Computing.

A CPU running at saturation is less of a problem than other resource types, as higher-priority work can preempt the current thread.

6.3.9 Preemption

Preemption, introduced in Chapter 3, Operating Systems, allows a higher-priority thread to preempt the currently running thread and begin its own execution instead. This eliminates the run-queue latency for higher-priority work, improving its performance.

6.3.10 Priority Inversion

Priority inversion occurs when a lower-priority thread holds a resource and blocks a higher-priority thread from running. This reduces the performance of the higher-priority work, as it is blocked waiting.

Solaris-based kernels implement a full *priority inheritance* scheme to avoid priority inversion. Here is an example of how this can work (based on a real-world case):

1. Thread A performs monitoring and has a low priority. It acquires an address space lock for a production database, to check memory usage.
2. Thread B, a routine task to perform compression of system logs, begins running.
3. There is insufficient CPU to run both. Thread B preempts A and runs.
4. Thread C is from the production database, has a high priority, and has been sleeping waiting for I/O. This I/O now completes, putting thread C back into the runnable state.
5. Thread C preempts B, runs, but then blocks on the address space lock held by thread A. Thread C leaves CPU.
6. The scheduler picks the next-highest-priority thread to run: B.
7. With thread B running, a high-priority thread, C, is effectively blocked on a lower-priority thread, B. This is priority inversion.
8. Priority inheritance gives thread A thread C's high priority, preempting B, until it releases the lock. Thread C can now run.

Linux since 2.6.18 has provided a user-level mutex that supports priority inheritance, intended for real-time workloads [1].

6.3.11 Multiprocess, Multithreading

Most processors provide multiple CPUs of some form. For an application to make use of them, it needs separate threads of execution so that it can run in parallel. For a 64-CPU system, for example, this may mean that an application can execute up to 64 times faster if it can make use of all CPUs in parallel, or handle 64 times the load. The degree to which the application can effectively scale with an increase in CPU count is a measure of *scalability*.

The two techniques to scale applications across CPUs are *multiprocess* and *multithreading*, which are pictured in Figure 6.4.

On Linux both the multiprocess and multithread models may be used, and both are implemented by tasks.

Differences between multiprocess and multithreading are shown in Table 6.1.

With all the advantages shown in the table, multithreading is generally considered superior, although more complicated for the developer to implement.

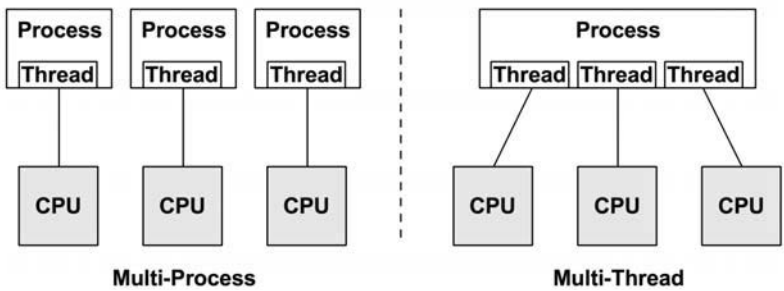


Figure 6-4 Software CPU scalability techniques

Table 6-1 Multiprocess and Multithreading Attributes

Attribute	Multiprocess	Multithreading
Development	Can be easier. Use of `fork()`.	Use of threads API.
Memory overhead	Separate address space per process consumes some memory resources.	Small. Requires only extra stack and register space.
CPU overhead	Cost of `fork()`/`exit()`, which includes MMU work to manage address spaces.	Small. API calls.
Communication	Via IPC. This incurs CPU cost including context switching for moving data between address spaces, unless shared memory regions are used.	Fastest. Direct access to share memory. Integrity via synchronization primitives (e.g., mutex locks).
Memory usage	While some memory may be duplicated, separate processes can `exit()` and return all memory back to the system.	Via system allocator. This may incur some CPU contention from multiple threads, and fragmentation before memory is reused.

Whichever technique is used, it is important that enough processes or threads be created to span the desired number of CPUs—which, for maximum performance, may be all of the CPUs available. Some applications may perform better when running on fewer CPUs, when the cost of thread synchronization and reduced memory locality outweighs the benefit of running across more CPUs.

Parallel architectures are also discussed in Chapter 5, Applications.

6.3.12 Word Size

Processors are designed around a maximum *word size*—32-bit or 64-bit—which is the integer size and register size. Word size is also commonly used, depending on

the processor, for the address space size and data path width (where it is sometimes called the *bit width*).

Larger sizes can mean better performance, although it's not as simple as it sounds. Larger sizes may cause memory overheads for unused bits in some data types. The data footprint also increases when the size of pointers (word size) increases, which can require more memory I/O. For the x86 64-bit architecture, these overheads are compensated by an increase in registers and a more efficient register calling convention, so 64-bit applications will more likely be faster than their 32-bit versions.

Processors and operating systems can support multiple word sizes and can run applications compiled for different word sizes simultaneously. If software has been compiled for the smaller word size, it may execute successfully but perform relatively poorly.

6.3.13 Compiler Optimization

The CPU runtime of applications can be significantly improved through compiler options (including setting word size) and optimizations. Compilers are also frequently updated to take advantage of the latest CPU instruction sets and to implement other optimizations. Sometimes application performance can be significantly improved simply by using a newer compiler.

This topic is covered in more detail in Chapter 5, Applications.

6.4 Architecture

This section introduces CPU architecture and implementation, for both hardware and software. Simple CPU models were introduced in Section 6.2, Models, and generic concepts in the previous section.

These topics have been summarized as background for performance analysis. For more details, see vendor processor manuals and texts on operating system internals. Some are listed at the end of this chapter.

6.4.1 Hardware

CPU hardware includes the processor and its subsystems, and the CPU interconnect for multiprocessor systems.

Processor

Components of a generic two-core processor are shown in Figure 6.5.

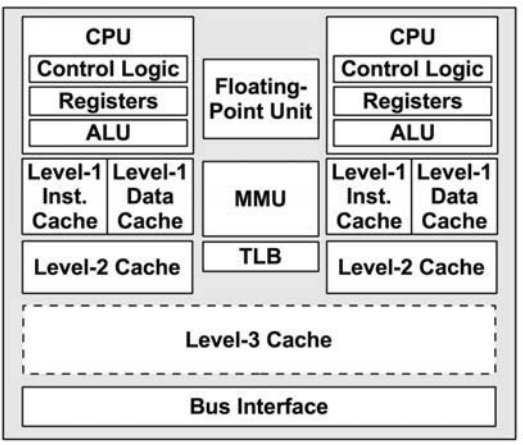


Figure 6-5 Generic two-core processor components

The *control unit* (pictured as *control logic*) is the heart of the CPU, performing instruction fetch, decoding, managing execution, and storing results.

This example processor depicts a shared floating-point unit and (optional) shared Level 3 cache. The actual components in your processor will vary depending on its type and model. Other performance-related components that may be present include the following:

- **P-cache:** prefetch cache (per CPU)
- **W-cache:** write cache (per CPU)
- **Clock:** signal generator for the CPU clock (or provided externally)
- **Timestamp counter:** for high-resolution time, incremented by the clock
- **Microcode ROM:** quickly converts instructions to circuit signals
- **Temperature sensors:** for thermal monitoring
- **Network interfaces:** if present on-chip (for high performance)

Some processor types use the temperature sensors as input for dynamic overclocking of individual cores (including Intel Turbo Boost technology), improving performance while the core remains in its temperature envelope.

CPU Caches

Various hardware caches are usually included in the processor (referred to as *on-chip*, *on-die*, *embedded*, or *integrated*) or with the processor (*external*). These improve memory performance by using faster memory types for caching reads and

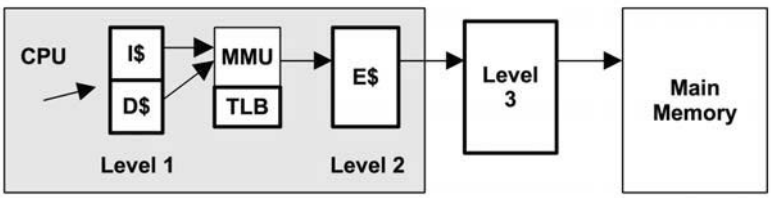


Figure 6-6 CPU cache hierarchy

buffering writes. The levels of cache access for a generic processor are shown in Figure 6.6.

They include

- **Level 1 instruction cache** (I$)
- **Level 1 data cache** (D$)
- **Translation lookaside buffer** (TLB)
- **Level 2 cache** (E$)
- **Level 3 cache** (optional)

The *E* in E$ originally stood for *external* cache, but with the integration of Level 2 caches it has since been cleverly referred to as *embedded* cache. The "Level" terminology is used nowadays instead of the "E$"-style notation, which avoids such confusion.

The caches available on each processor depend on its type and model. Over time, the number and sizes of these caches have been increasing. This is illustrated in Table 6.2 by the listing of Intel processors since 1978, including advances in caches [Intel 12].

Table 6-2 Example Intel Processor Cache Sizes from 1978 to 2011

Processor	Date	Max Clock	Transistors	Data Bus	Level 1	Level 2	Level 3
8086	1978	8 MHz	29 K	16-bit	—		
Intel 286	1982	12.5 MHz	134 K	16-bit	—		
Intel 386 DX	1985	20 MHz	275 K	32-bit	—	—	—
Intel 486 DX	1989	25 MHz	1.2 M	32-bit	8 KB	—	—
Pentium	1993	60 MHz	3.1 M	64-bit	16 KB	—	—
Pentium Pro	1995	200 MHz	5.5 M	64-bit	16 KB	256/512 KB	—

continues

Table 6-2 Example Intel Processor Cache Sizes from 1978 to 2011 (*Continued*)

Processor	Date	Max Clock	Transistors	Data Bus	Level 1	Level 2	Level 3
Pentium II	1997	266 MHz	7 M	64-bit	32 KB	256/512 KB	—
Pentium III	1999	500 MHz	8.2 M	64-bit	32 KB	512 KB	—
Intel Xeon	2001	1.7 GHz	42 M	64-bit	8 KB	512 KB	—
Pentium M	2003	1.6 GHz	77 M	64-bit	64 KB	1 MB	—
Intel Xeon MP	2005	3.33 GHz	675 M	64-bit	16 KB	1 MB	8 MB
Intel Xeon 7410	2006	3.4 GHz	1.3 B	64-bit	64 KB	2 x 1 MB	16 MB
Intel Xeon 7460	2008	2.67 GHz	1.9 B	64-bit	64 KB	3 x 3 MB	16 MB
Intel Xeon 7560	2010	2.26 GHz	2.3 B	64-bit	64 KB	256 KB	24 MB
Intel Xeon E7-8870	2011	2.4 GHz	2.2 B	64-bit	64 KB	256 KB	30 MB

For multicore and multithreading processors, some of these caches may be shared between cores and threads.

Apart from the increasing number and sizes of CPU caches, there is also a trend toward providing these on-chip, where access latency can be minimized, instead of providing them externally to the processor.

Latency

Multiple levels of cache are used to deliver the optimum configuration of size and latency. The access time for the Level 1 cache is typically a few CPU clock cycles, and for the larger Level 2 cache around a dozen clock cycles. Main memory can take around 60 ns (around 240 cycles, for a 4 GHz processor), and address translation by the MMU also adds latency.

The CPU cache latency characteristics for your processor can be determined experimentally using micro-benchmarking [Ruggiero 08]. Figure 6.7 shows the result of this, plotting memory access latency for an Intel Xeon E5620 2.4 GHz tested over increasing ranges of memory using LMbench [2].

Both axes are logarithmic. The steps in the graphs show when a cache level was exceeded, and access latency becomes a result of the next (slower) cache level.

Associativity

Associativity is a cache characteristic describing a constraint for locating new entries in the cache. Types are

- **Fully associative:** The cache can locate new entries anywhere. For example, an LRU algorithm could evict the least recently used entry in the entire cache.

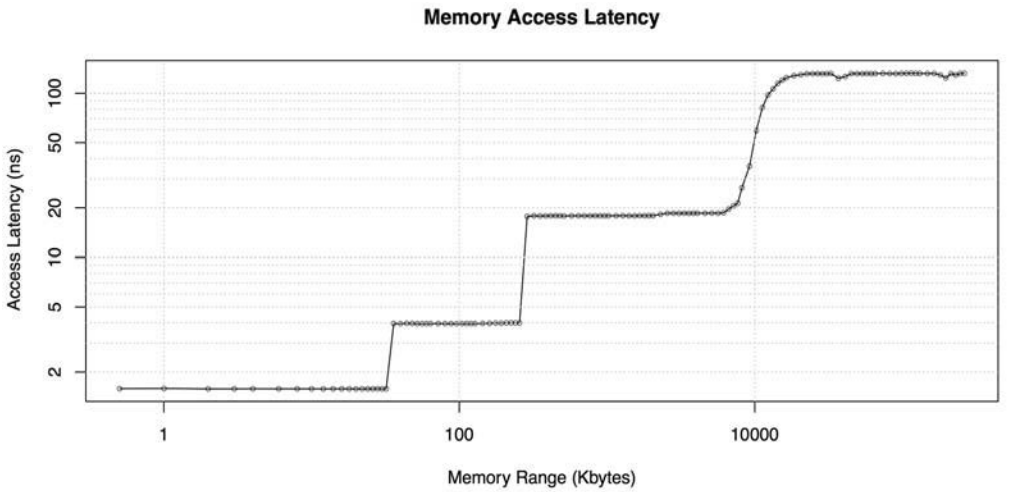


Figure 6-7 Memory access latency testing

- **Direct mapped:** Each entry has only one valid location in the cache, for example, a hash of the memory address, using a subset of the address bits to form an address in the cache.
- **Set associative:** A subset of the cache is identified by mapping (e.g., hashing), from within which another algorithm (e.g., LRU) may be performed. It is described in terms of the subset size; for example, *four-way set associative* maps an address to four possible locations, and then picks the best from those four.

CPU caches often use set associativity as a balance between fully associative (which is expensive to perform) and direct mapped (which has poor hit rates).

Cache Line

Another characteristic of CPU caches is their *cache line* size. This is a range of bytes that are stored and transferred as a unit, improving memory throughput. A typical cache line size for x86 processors is 64 bytes. Compilers take this into account when optimizing for performance. Programmers sometimes do as well; see Hash Tables in Section 5.2.5 of Chapter 5, Applications.

Cache Coherency

Memory may be cached in multiple CPU caches on different processors at the same time. When one CPU modifies memory, all caches need to be aware that their cached copy is now *stale* and should be discarded, so that any future reads will retrieve the newly modified copy. This process, called *cache coherency*, ensures that

CPUs are always accessing the correct state of memory. It is also one of the greatest challenges when designing scalable multiprocessor systems, as memory can be modified rapidly.

MMU

The MMU is responsible for virtual-to-physical address translation. A generic MMU is pictured in Figure 6.8, along with CPU cache types. This MMU uses an on-chip TLB to cache address translations. Cache misses are satisfied by translation tables in main memory (DRAM), called *page tables*, which are read directly by the MMU (hardware).

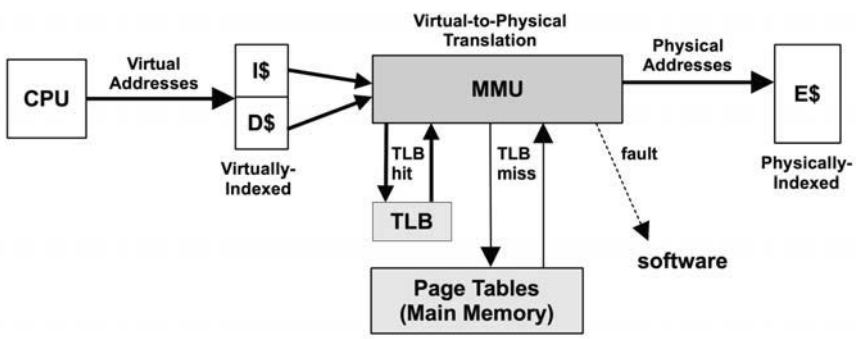


Figure 6-8 Memory management unit and CPU caches

These factors are processor-dependent. Some (older) processors handle TLB misses using software to walk the page tables, and then populate the TLB with the requested mappings. Such software may maintain its own, larger, in-memory cache of translations, called the *translation storage buffer* (TSB). Newer processors can service TLB misses in hardware, greatly reducing their cost.

Interconnects

For multiprocessor architectures, processors are connected using either a shared system bus or a dedicated interconnect. This is related to the memory architecture of the system, uniform memory access (UMA) or NUMA, as discussed in Chapter 7, Memory.

A shared system bus, called the *front-side bus*, used by earlier Intel processors is illustrated by the four-processor example in Figure 6.9.

The use of a system bus has scalability problems when the processor count is increased, due to contention for the shared bus resource. Modern servers are typically multiprocessor, NUMA, and use a CPU interconnect instead.

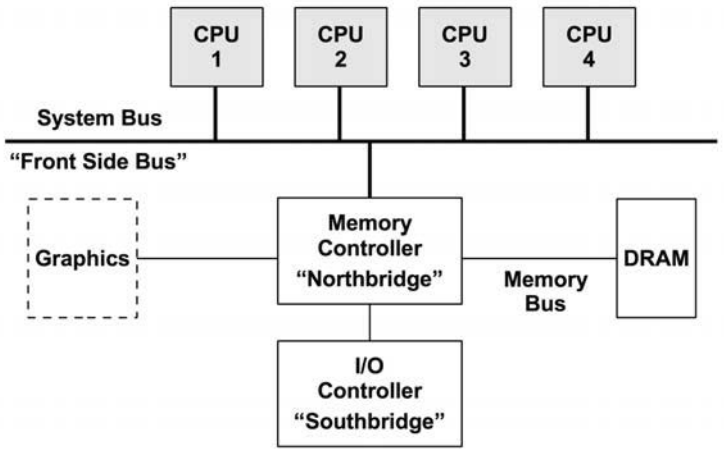


Figure 6-9 Example Intel front-side bus architecture, four-processor

Interconnects can connect components other than processors, such as I/O controllers. Example interconnects include Intel's Quick Path Interconnect (QPI) and AMD's HyperTransport (HT). An example Intel QPI architecture for a four-processor system is shown in Figure 6.10.

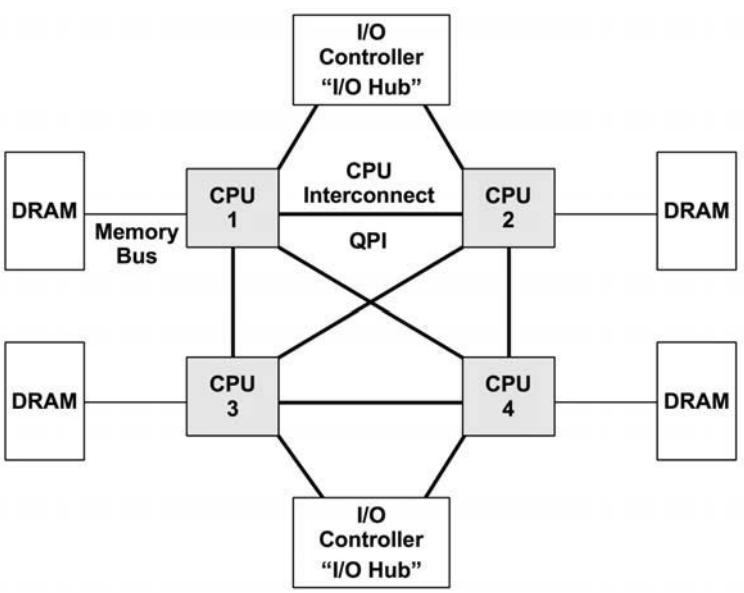


Figure 6-10 Example Intel QPI architecture, four-processor

The private connections between processors allow for noncontended access and also allow higher bandwidths than the shared system bus. Some example speeds for Intel FSB and QPI are shown in Table 6.3 [Intel 09].

Table 6-3 Intel CPU Interconnect Bandwidths

Intel	Transfer Rate	Width	Bandwidth
FSB (2007)	1.6 GT/s	8 bytes	12.8 Gbytes/s
QPI (2008)	6.4 GT/s	2 bytes	25.6 Gbytes/s

QPI is *double-pumped*, performing a data transfer on both edges of the clock, doubling the data transfer rate. This explains the bandwidth shown in the table (6.4 GT/s x 2 bytes x double = 25.6 Gbytes/s).

Apart from external interconnects, processors have internal interconnects for core communication.

Interconnects are typically designed for high bandwidth, so that they do not become a systemic bottleneck. If they do, performance will degrade as CPU instructions encounter stall cycles for operations that involve the interconnect, such as remote memory I/O. A key indicator for this is a rise in CPI. CPU instructions, cycles, CPI, stall cycles, and memory I/O can be analyzed using CPU performance counters.

CPU Performance Counters

CPU performance counters (CPCs) go by many names, including *performance instrumentation counters* (PICs), *performance monitoring unit* (PMU), *hardware events*, and *performance monitoring events*. They are processor registers that can be programmed to count low-level CPU activity. They typically include counters for the following:

- **CPU cycles:** including stall cycles and types of stall cycles
- **CPU instructions:** retired (executed)
- **Level 1, 2, 3 cache accesses:** hits, misses
- **Floating-point unit:** operations
- **Memory I/O:** reads, writes, stall cycles
- **Resource I/O:** reads, writes, stall cycles

Each CPU has a small number of registers, usually between two and eight, that can be programmed to record events like these. Those available depend on the processor type and model and are documented in the processor manual.

As a relatively simple example, the Intel P6 family of processors provide performance counters via four model-specific registers (MSRs). Two MSRs are the counters and are read-only. The other two MSRs are used to program the counters, called *event-select* MSRs, and are read-write. The performance counters are 40-bit registers, and the event-select MSRs are 32-bit. The format of the event-select MSRs is shown in Figure 6.11.

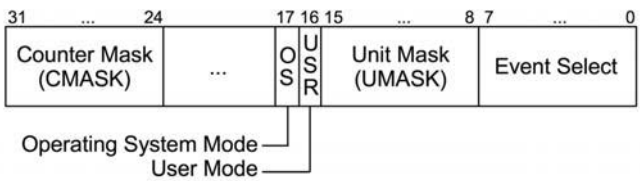


Figure 6-11 Example Intel performance event-select MSR

The counter is identified by the event select and the UMASK. The event select identifies the type of event to count, and the UMASK identifies subtypes or groups of subtypes. The OS and USR bits can be set so that the counter is incremented only while in kernel mode (OS) or user mode (USR), based on the processor protection rings. The CMASK can be set to a threshold of events that must be reached before the counter is incremented.

The Intel processor manual (volume 3B [Intel 13]) lists the dozens of events that can be counted by their event-select and UMASK values. The selected examples in Table 6.4 provide an *idea* of the different targets (processor functional units) that may be observable. You will need to refer to your current processor manual to see what you actually have.

There are many, many more counters, especially for newer processors. The Intel Sandy Bridge family of processors provide not only more counter types, but also more counter registers: three fixed and four programmable counters per hardware thread, and an additional eight programmable counters per core ("general-purpose"). These are 48-bit counters when read.

Since performance counters vary among manufacturers, a standard has been developed to provide a consistent interface across them. This is the *Processor Application Programmers Interface* (PAPI). Instead of the Intel names seen in Table 6.4, PAPI assigns generic names to the counter types, for example, PAPI_tot_cyc for total cycle counts, instead of CPU_CLK_UNHALTED.

Table 6-4 Selected Examples of Intel CPU Performance Counters

Event Select	UMASK	Unit	Name	Description
0x43	0x00	data cache	DATA_MEM_ REFS	All loads from any memory type. All stores to any memory type. Each part of a split is counted separately. . . . Does not include I/O accesses or other nonmemory accesses.
0x48	0x00	data cache	DCU_MISS_ OUTSTANDING	Weighted number of cycles while a DCU miss is outstanding, incremented by the number of outstanding cache misses at any particular time. Cacheable read requests only are considered. . . .
0x80	0x00	instruction fetch unit	IFU_IFETCH	Number of instruction fetches, both cacheable and noncacheable, including UC (uncacheable) fetches.
0x28	0x0F	L2 cache	L2_IFETCH	Number of L2 instruction fetches. . . .
0xC1	0x00	floating-point unit	FLOPS	Number of computational floating-point operations retired. . . .
0x7E	0x00	external bus logic	BUS_SNOOP_ STALL	Number of clock cycles during which the bus is snoop stalled.
0xC0	0x00	instruction decoding and retirement	INST_RETIRED	Number of instructions retired.
0xC8	0x00	interrupts	HW_INT_RX	Number of hardware interrupts received.
0xC5	0x00	branches	BR_MISS_PRED_ RETIRED	Number of mispredicted branches retired.
0xA2	0x00	stalls	RESOURCE_ STALLS	Incremented by one during every cycle for which there is a resource-related stall. . . .
0x79	0x00	clocks	CPU_CLK_ UNHALTED	Number of cycles during which the processor is not halted.

6.4.2 Software

Kernel software to support CPUs includes the scheduler, scheduling classes, and the idle thread.

Scheduler

Key functions of the kernel CPU scheduler are shown in Figure 6.12.

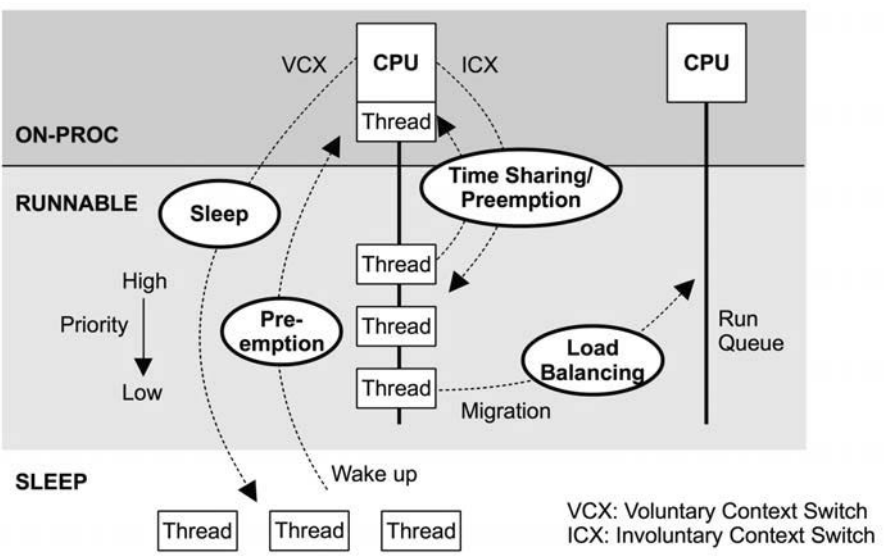


Figure 6-12 Kernel CPU scheduler functions

These are

- **Time sharing:** multitasking between runnable threads, executing those with the highest priority first.
- **Preemption:** For threads that have become runnable at a high priority, the scheduler can preempt the currently running thread, so that execution of the higher-priority thread can begin immediately.
- **Load balancing:** moving runnable threads to the run queues of idle or less busy CPUs.

The figure shows run queues per CPU. There are also run queues per priority level, so that the scheduler can easily manage which thread of the same priority should run.

A brief summary of how scheduling works for recent Linux and Solaris-based kernels follows. Function names are included, so that you can find them in the source code for further reference (although they may have changed). Also refer to internals texts, listed in the Bibliography.

Linux

On Linux, time sharing is driven by the system timer interrupt by calling `scheduler_tick()`, which calls scheduler class functions to manage priorities and the expiry of units of CPU time called *time slices*. Preemption is triggered when threads become runnable and the scheduler class `check_preempt_curr()` function is called. Switching of threads is managed by `__schedule()`, which selects the highest-priority thread via `pick_next_task()` for running. Load balancing is performed by the `load_balance()` function.

Solaris

On Solaris-based kernels, time sharing is driven by `clock()`, which calls scheduler class functions including `ts_tick()` to check for time slice expiration. If the thread has exceeded its time, its priority is reduced, allowing another thread to preempt. Preemption is handled by `preempt()` for user threads and `kpreempt()` for kernel threads. The `swtch()` function manages a thread leaving CPU for any reason, including from voluntary context switching, and calls dispatcher functions to find the best runnable thread to take its place: `disp()`, `disp_getwork()`, or `disp_getbest()`. Load balancing includes the idle thread calling similar functions to find runnable threads from another CPU's dispatcher queue (run queue).

Scheduling Classes

Scheduling classes manage the behavior of runnable threads, specifically their priorities, whether their on-CPU time is *time-sliced*, and the duration of those *time slices* (also known as *time quantum*). There are also additional controls via scheduling *policies*, which may be selected within a scheduling class and can control scheduling between threads of the same priority. Figure 6.13 depicts them along with the thread priority range.

The priority of user-level threads is affected by a user-defined *nice* value, which can be set to lower the priority of unimportant work. In Linux, the nice value sets the *static priority* of the thread, which is separate from the *dynamic priority* that the scheduler calculates.

Note that the priority ranges are inverted between Linux and Solaris-based kernels. The original Unix priority range (6th edition) used lower numbers for higher priority, the system Linux uses now.

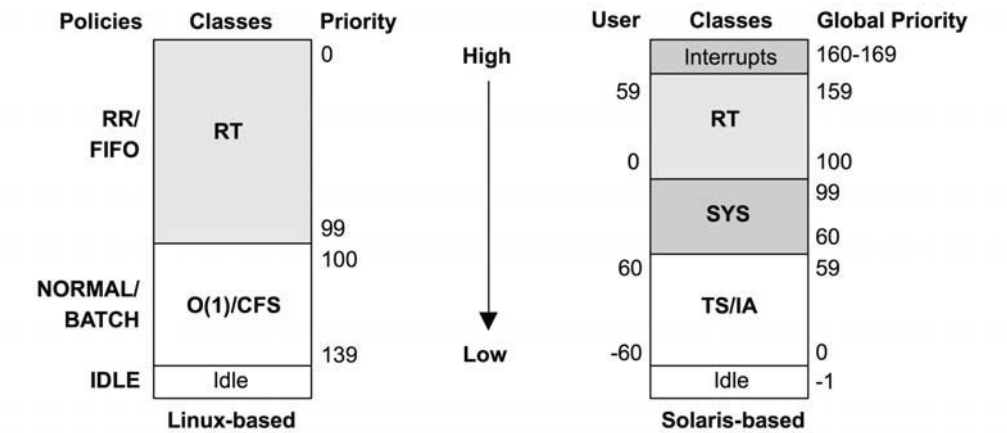


Figure 6-13 Thread scheduler priorities

Linux

For Linux kernels, the scheduling classes are

- **RT:** provides fixed and high priorities for real-time workloads. The kernel supports both user- and kernel-level preemption, allowing RT tasks to be dispatched with low latency. The priority range is 0–99 (MAX_RT_PRIO–1).
- **O(1):** The O(1) scheduler was introduced in Linux 2.6 as the default time-sharing scheduler for user processes. The name comes from the algorithm complexity of O(1) (see Chapter 5, Applications, for a summary of big O notation). The prior scheduler contained routines that iterated over all tasks, making it O(n), which became a scalability issue. The O(1) scheduler dynamically improves the priority of I/O-bound over CPU-bound workloads, to reduce latency of interactive and I/O workloads.
- **CFS:** Completely fair scheduling was added to the Linux 2.6.23 kernel as the default time-sharing scheduler for user processes. The scheduler manages tasks on a red-black tree instead of traditional run queues, which is keyed from the task CPU time. This allows low CPU consumers to be easily found and executed in preference to CPU-bound workloads, improving the performance of interactive and I/O-bound workloads.

The scheduling class behavior can be adjusted by user-level processes by calling `sched_setscheduler()` to set the scheduler policy. The RT class supports the SCHED_RR and SCHED_FIFO policies, and the CFS class supports SCHED_NORMAL and SCHED_BATCH.

Scheduler policies are as follows:

- **RR:** SCHED_RR is round-robin scheduling. Once a thread has used its time quantum, it is moved to the end of the run queue for that priority level, allowing others of the same priority to run.
- **FIFO:** SCHED_FIFO is first-in first-out scheduling, which continues running the thread at the head of the run queue until it voluntarily leaves, or until a higher-priority thread arrives. The thread continues to run, even if other threads of the same priority are on the run queue.
- **NORMAL:** SCHED_NORMAL (previously known as SCHED_OTHER) is time-sharing scheduling and is the default for user processes. The scheduler dynamically adjusts priority based on the scheduling class. For O(1), the time slice duration is set based on the static priority: longer durations for higher-priority work. For CFS, the time slice is dynamic.
- **BATCH:** SCHED_BATCH is similar to SCHED_NORMAL, but with the expectation that the thread will be CPU-bound and should not be scheduled to interrupt other I/O-bound interactive work.

Other classes and policies may be added over time. Scheduling algorithms have been researched that are *hyperthreading-aware* [Bulpin 05] and *temperature-aware* [Otto 06], which optimize performance by accounting for additional processor factors.

When there is no thread to run, a special *idle task* (also called *idle thread*) is executed as a placeholder until another thread is runnable.

Solaris

For Solaris-based kernels, the scheduling classes are as follows:

- **RT:** Real-time scheduling provides fixed and high priorities for real-time workloads. These preempt all other work (except interrupt service routines) so that application response time can be deterministic—a typical requirement for real-time workloads.
- **SYS:** System is a high-priority scheduling class for kernel threads. These threads have a fixed priority and execute for as long as needed (or until preempted by RT or interrupts).
- **TS:** Time sharing is the default for user processes; it dynamically adjusts priority and quantum based on recent CPU usage. Thread priority is demoted if it uses its quantum, and the quantum is increased. This causes CPU-bound workloads to run at a low priority with large time quantums (reducing

scheduler costs), and I/O-bound workloads—which voluntarily context switch before their quantum is used—to run at a high priority. The result is that the performance of I/O-bound workloads is not affected by the presence of long-running CPU jobs. This class also applies the nice value, if set.

- **IA:** Interactive is similar to TS, but with a slightly higher default priority. It is rarely used today (it was previously used to improve the responsiveness of graphical X sessions).
- **FX:** Fixed (not pictured in Figure 6.13) is a process scheduling class for setting fixed priorities, in the same global priority range as TS (0–59).
- **FSS:** Fair-share scheduling (not pictured in Figure 6.13) manages CPU usage between groups of processes, either *projects* or *zones*, based on share values. This allows groups of projects to use the CPUs fairly based on shares, instead of based on their number of threads or processes. Each process group can consume a fraction of CPU calculated from its share value divided by the total *busy* shares on the system at that time. This means that if that group is the only busy group, it can use all CPU resources. FSS is in popular use for cloud computing, so that tenants (zones) can be allocated shares fairly and can also consume more CPU if it is available and unused. FSS exists in the same global priority range as TS (0–59) and has a fixed time quantum.
- **SYSDC:** The system duty cycle scheduling class is for kernel threads that are large CPU consumers, such as the ZFS transaction group flush thread. It allows a target duty cycle to be specified (the ratio of CPU time to runnable time) and will deschedule the thread to match the duty cycle. This prevents long-running kernel threads, which would otherwise be in the SYS class, from starving other threads that need to use that CPU.
- **Interrupts:** For the purpose of scheduling interrupt threads, they are given a priority that is 159 + IPL (see Section 3.2.3, Interrupts and Interrupt Threads, in Chapter 3, Operating Systems).

Solaris-based systems also support scheduling policies (not pictured in Figure 6.13) that are set using `sched_setscheduler()`: SCHED_FIFO, SCHED_RR, and SCHED_OTHER (time sharing).

The idle thread is a special case, running with the lowest priority.

Idle Thread

The kernel "idle" thread (or *idle task*) runs on-CPU when there is no other runnable thread and has the lowest possible priority. It is usually programmed to inform the processor that CPU execution may either be halted (halt instruction) or throttled down to conserve power. The CPU will wake up on the next hardware interrupt.

NUMA Grouping

Performance on NUMA systems can be significantly improved by making the kernel *NUMA-aware*, so that it can make better scheduling and memory placement decisions. This can automatically detect and create groups of localized CPU and memory resources and organize them in a topology to reflect the NUMA architecture. This topology allows the cost of any memory access to be estimated.

On **Linux** systems, these are called *scheduling domains* [3], which are in a topology beginning with the *root domain*.

On **Solaris**-based systems, these are called *locality groups* (lgrps) and begin with the *root group*.

A manual form of grouping can be performed by the system administrator, either by binding processes to run on one or more CPUs only, or by creating an exclusive set of CPUs for processes to run on. See Section 6.5.10, CPU Binding.

Processor Resource-Aware

Other than for NUMA, the CPU resource topology can be understood by the kernel so that it can make better scheduling decisions for power management and load balancing. On Solaris-based systems, this is implemented by *processor groups*.

6.5 Methodology

This section describes various methodologies and exercises for CPU analysis and tuning. Table 6.5 summarizes the topics.

Table 6-5 CPU Performance Methodologies

Methodology	Types
Tools method	observational analysis
USE method	observational analysis
Workload characterization	observational analysis, capacity planning
Profiling	observational analysis
Cycle analysis	observational analysis
Performance monitoring	observational analysis, capacity planning
Static performance tuning	observational analysis, capacity planning
Priority tuning	tuning
Resource controls	tuning
CPU binding	tuning

Table 6-5 CPU Performance Methodologies (*Continued*)

Methodology	Types
Micro-benchmarking	experimental analysis
Scaling	capacity planning, tuning

See Chapter 2, Methodology, for more strategies and the introduction to many of these. You are not expected to use them all; treat this as a cookbook of recipes that may be followed individually or used in combination.

My suggestion is to use the following, in this order: performance monitoring, the USE method, profiling, micro-benchmarking, and static analysis.

Section 6.6, Analysis, shows operating system tools for applying these strategies.

6.5.1 Tools Method

The tools method is a process of iterating over available tools, examining key metrics they provide. While this is a simple methodology, it can overlook issues for which the tools provide poor or no visibility, and it can be time-consuming to perform.

For CPUs, the tools method can involve checking the following:

- **`uptime:`** Check load averages to see if CPU load is increasing or decreasing over time. A load average over the number of CPUs in the system usually indicates saturation.
- **`vmstat:`** Run `vmstat` per second, and check the idle columns to see how much headroom there is. Less than 10% can be a problem.
- **`mpstat:`** Check for individual hot (busy) CPUs, identifying a possible thread scalability problem.
- **`top/prstat:`** See which processes and users are the top CPU consumers.
- **`pidstat/prstat:`** Break down the top CPU consumers into user- and system-time.
- **`perf/dtrace/stap/oprofile:`** Profile CPU usage stack traces for either user- or kernel-time, to identify why the CPUs are in use.
- **`perf/cpustat:`** Measure CPI.

If an issue is found, examine all fields from the available tools to learn more context. See Section 6.6, Analysis, for more about each tool.

6.5.2 USE Method

The USE method is for identifying bottlenecks and errors across all components, early in a performance investigation, before deeper and more time-consuming strategies are followed.

For each CPU, check for

- **Utilization:** the time the CPU was busy (not in the idle thread)
- **Saturation:** the degree to which runnable threads are queued waiting their turn on-CPU
- **Errors:** CPU errors, including correctable errors

Errors may be checked first since they are typically quick to check and the easiest to interpret. Some processors and operating systems will sense an increase in correctable errors (error-correcting code, ECC) and will offline a CPU as a precaution, before an uncorrectable error causes a CPU failure. Checking for these errors can be a matter of checking that all CPUs are still online.

Utilization is usually readily available from operating system tools as *percent busy*. This metric should be examined per CPU, to check for scalability issues. It can also be examined per core, for cases where a core's resources are heavily utilized, preventing idle hardware threads from executing. High CPU and core utilization can be understood by using profiling and cycle analysis.

For environments that implement CPU limits or quotas (resource controls), as occurs in some cloud computing environments, CPU utilization may need to be measured in terms of the imposed limit, in addition to the physical limit. Your system may exhaust its CPU quota well before the physical CPUs reach 100% utilization, encountering saturation earlier than expected.

Saturation metrics are commonly provided system-wide, including as part of load averages. This metric quantifies the degree to which the CPUs are overloaded, or a CPU quota, if present, is used up.

6.5.3 Workload Characterization

Characterizing the load applied is important in capacity planning, benchmarking, and simulating workloads. It can also lead to some of the largest performance gains by identifying unnecessary work that can be eliminated.

Basic attributes for characterizing CPU workload are

- Load averages (utilization + saturation)
- User-time to system-time ratio

- Syscall rate
- Voluntary context switch rate
- Interrupt rate

The intent is to characterize the applied load, not the delivered performance. The load average is suited for this, as it reflects the CPU load requested, regardless of the delivered performance as shown by the utilization/saturation breakdown. See the example and further explanation in Section 6.6.1, uptime.

The rate metrics are a little harder to interpret, as they reflect both the applied load and to some degree the delivered performance, which can throttle their rate.

The user-time to system-time ratio shows the type of load applied, as introduced earlier in Section 6.3.7, User-Time/Kernel-Time. High user-time rates are due to applications spending time performing their own compute. High system-time shows time spent in the kernel instead, which may be further understood by the syscall and interrupt rate. I/O-bound workloads have higher system-time, syscalls, and also voluntary context switches as threads block waiting for I/O.

Here is an example workload description that you might receive, designed to show how these attributes can be expressed together:

> On our busiest application server, the load average varies between 2 and 8 during the day depending on the number of active clients. The user/system ratio is 60/40, as this is an I/O-intensive workload performing around 100 K syscalls/s, and a high rate of voluntary context switches.

These characteristics can vary over time as different load is encountered.

Advanced Workload Characterization/Checklist

Additional details may be included to characterize the workload. These are listed here as questions for consideration, which may also serve as a checklist when studying CPU issues thoroughly:

- What is the CPU utilization system-wide? Per CPU?
- How parallel is the CPU load? Is it single-threaded? How many threads?
- Which applications or users are using the CPUs? How much?
- Which kernel threads are using the CPUs? How much?
- What is the CPU usage of interrupts?
- What is the CPU interconnect utilization?
- Why are the CPUs being used (user- and kernel-level call paths)?
- What types of stall cycles are encountered?

See Chapter 2, Methodology, for a higher-level summary of this methodology and the characteristics to measure (who, why, what, how). The sections that follow expand upon the last two questions in this list: how call paths can be analyzed using profiling, and stall cycles using cycle analysis.

6.5.4 Profiling

Profiling builds a picture of the target for study. CPU usage can be profiled by sampling the state of the CPUs at timed intervals, following these steps:

1. **Select** the type of profile data to capture, and the rate.
2. **Begin** sampling at a timed interval.
3. **Wait** while the activity of interest occurs.
4. **End** sampling and collect sample data.
5. **Process** the data.

Some profiling tools, including DTrace, allow real-time processing of the captured data, which can be analyzed while sampling is still occurring.

Processing and navigating the data may be enhanced by a separate toolset from the one used to collect the data. One example is flame graphs (covered later), which process the output of DTrace and other profiling tools. Another is the Performance Analyzer from Oracle Solaris Studio, which automates collecting and browsing the profile data with the target source code.

The types of CPU profile data are based on the following factors:

- User level, kernel level, or both
- Function and offset (program-counter-based), function only, partial stack trace, or full stack trace

Selecting full stack traces for both user and kernel level captures the complete profile of CPU usage. However, it typically generates an excessive amount of data. Capturing only user or kernel, partial stacks (e.g., five levels deep), or even just the executing function name may prove sufficient for identifying CPU usage from much less data.

As a simple example of profiling, the following DTrace one-liner samples the user-level function name at 997 Hz for a duration of 10 s:

```
# dtrace -n 'profile-997 /arg1 && execname == "beam.smp"/ {
    @[ufunc(arg1)] = count(); } tick-10s { exit(0); }'
[...]
```

```
libc.so.1`mutex_lock_impl                                29
libc.so.1`atomic_swap_8                                  33
beam.smp`make_hash                                       45
libc.so.1`__time                                         71
innostore_drv.so`os_aio_array_get_nth_slot               80
beam.smp`process_main                                   127
libc.so.1`mutex_trylock_adaptive                        140
innostore_drv.so`os_aio_simulated_handle                158
beam.smp`sched_sys_wait                                 202
libc.so.1`memcpy                                        258
innostore_drv.so`ut_fold_binary                        1800
innostore_drv.so`ut_fold_ulint_pair                    4039
```

DTrace has already performed step 5, processing the data by aggregating function names and printing the sorted frequency counts. This shows that the most common on-CPU user-level function while tracing was `ut_fold_ulint_pair()`, which was sampled 4,039 times.

A frequency of 997 Hz was used to avoid sampling in lockstep with any activity (e.g., timed tasks running at 100 or 1,000 Hz).

By sampling the full stack trace, the code path for CPU usage can be identified, which typically points to higher-level reasons for CPU usage. More examples of sampling are given in Section 6.6, Analysis. Also see Chapter 5, Applications, for more on CPU profiling, including fetching other programming language context from the stack.

For the usage of specific CPU resources, such as caches and interconnects, profiling can use CPC-based event triggers instead of timed intervals. This is described in the next section on cycle analysis.

6.5.5 Cycle Analysis

By using the CPU performance counters (CPCs), CPU utilization can be understood at the cycle level. This may reveal that cycles are spent stalled on Level 1, 2, or 3 cache misses, memory I/O, or resource I/O, or spent on floating-point operations or other activity. This information may lead to performance wins by adjusting compiler options or changing the code.

Begin cycle analysis by measuring CPI. If CPI is high, continue to investigate types of stall cycles. If CPI is low, look for ways in the code to reduce instructions performed. The values for "high" or "low" CPI depend on your processor: low could be less than one, and high could be greater than ten. You can get a sense of these values by performing known workloads that are either memory-I/O-intensive or instruction-intensive and measuring the resulting CPI for each.

Apart from measuring counter values, CPC can be configured to interrupt the kernel on the overflow of a given value. For example, at every 10,000 Level 2 cache misses, the kernel could be interrupted to gather a stack backtrace. Over time, the

kernel builds a profile of the code paths that are causing Level 2 cache misses, without the prohibitive overhead of measuring every single miss. This is typically used by integrated developer environment (IDE) software, to annotate code with the locations that are causing memory I/O and stall cycles. Similar observability is possible using DTrace and the cpc provider.

Cycle analysis is an advanced activity that can take days to perform with command-line tools, as demonstrated in Section 6.6, Analysis. You should also expect to spend some quality time with your CPU vendor's processor manuals. Performance analyzers such as Oracle Solaris Studio can save time as they are programmed to find the CPCs of interest to you.

6.5.6 Performance Monitoring

Performance monitoring can identify active issues and patterns of behavior over time. Key metrics for CPUs are

- **Utilization:** percent busy
- **Saturation:** either run-queue length, inferred from load average, or as a measure of thread scheduler latency

Utilization should be monitored on a per-CPU basis to identify thread scalability issues. For environments that implement CPU limits or quotas (resource controls), such as some cloud computing environments, CPU usage compared to these limits also needs to be recorded.

A challenge when monitoring CPU usage is choosing an interval to measure and archive. Some monitoring tools use 5 minutes, which can hide the existence of shorter bursts of CPU utilization. Per-second measurements are preferable, but you should be aware that there can be bursts even within a second. These can be identified from saturation.

6.5.7 Static Performance Tuning

Static performance tuning focuses on issues of the configured environment. For CPU performance, examine the following aspects of the static configuration:

- How many CPUs are available for use? Are they cores? Hardware threads?
- Is the CPU architecture single- or multiprocessor?
- What is the size of the CPU caches? Are they shared?

- What is the CPU clock speed? Is it dynamic (e.g., Intel Turbo Boost and SpeedStep)? Are those dynamic features enabled in the BIOS?
- What other CPU-related features are enabled or disabled in the BIOS?
- Are there performance issues (bugs) with this processor model? Are they listed in the processor errata sheet?
- Are there performance issues (bugs) with this BIOS firmware version?
- Are there software-imposed CPU usage limits (resource controls) present? What are they?

The answers to these questions may reveal previously overlooked configuration choices.

The last question is especially true for cloud computing environments, where CPU usage is commonly limited.

6.5.8 Priority Tuning

Unix has always provided a `nice()` system call for adjusting process priority, which sets a nice-ness value. Positive nice values result in lower process priority (nicer), and negative values—which can be set only by the superuser (root)—result in higher priority. A `nice(1)` command became available to launch programs with nice values, and a `renice(1M)` command was later added (in BSD) to adjust the nice value of already running processes. The man page from Unix 4th edition provides this example [4]:

> The value of 16 is recommended to users who wish to execute long-running programs without flak from the administration.

The nice value is still useful today for adjusting process priority. This is most effective when there is contention for CPUs, causing scheduler latency for high-priority work. Your task is to identify low-priority work, which may include monitoring agents and scheduled backups, that can be modified to start with a nice value. Analysis may also be performed to check that the tuning is effective, and that the scheduler latency remains low for high-priority work.

Beyond nice, the operating system may provide more advanced controls for process priority such as changing the scheduler class or scheduler policy, or changing the tuning of the class. Both Linux and Solaris-based kernels include the *real-time scheduling class*, which can allow processes to preempt all other work. While this can eliminate scheduler latency (other than for other real-time processes and interrupts), make sure you understand the consequences. If the real-time application

encounters a bug where multiple threads enter an infinite loop, it can cause all CPUs to become unavailable for all other work—including the administrative shell required to manually fix the problem. This particular scenario is usually solved only by rebooting the system (oops!).

6.5.9 Resource Controls

The operating system may provide fine-grained controls for allocating CPU cycles to processes or groups of processes. These may include fixed limits for CPU utilization and shares for a more flexible approach—allowing idle CPU cycles to be consumed based on a share value. How these work is implementation-specific and discussed in Section 6.8, Tuning.

6.5.10 CPU Binding

Another way to tune CPU performance involves binding processes and threads to individual CPUs, or collections of CPUs. This can increase CPU cache warmth for the process, improving its memory I/O performance. For NUMA systems it also improves memory locality, also improving performance.

There are generally two ways this is performed:

- **Process binding:** configuring a process to run only on a single CPU, or only on one CPU from a defined set.
- **Exclusive CPU sets:** partitioning a set of CPUs that can be used only by the process(es) assigned to them. This can improve CPU cache further, as when the process is idle other processes cannot use the CPUs, leaving the caches warm.

On Linux-based systems, the exclusive CPU sets approach can be implemented using *cpusets*. On Solaris-based systems, this is called *processor sets*. Configuration examples are provided in Section 6.8, Tuning.

6.5.11 Micro-Benchmarking

There are various tools for CPU micro-benchmarking, which typically measure the time taken to perform a simple operation many times. The operation may be based on the following:

- **CPU instructions:** integer arithmetic, floating-point operations, memory loads and stores, branch and other instructions

- **Memory access:** to investigate latency of different CPU caches and main memory throughput
- **Higher-level languages:** similar to CPU instruction testing, but written in a higher-level interpreted or compiled language
- **Operating system operations:** testing system library and system call functions that are CPU-bound, such as `getpid()` and process creation

An early example of a CPU benchmark is Whetstone by the National Physical Laboratory, written in 1972 in Algol 60 and intended to simulate a scientific workload. The Dhrystone benchmark was later developed in 1984 to simulate integer workloads of the time and became a popular means to compare CPU performance. These, and various Unix benchmarks including process creation and pipe throughput, were included in a collection called *UnixBench*, originally from Monash University and published by *BYTE* magazine [Hinnant 84]. More recent CPU benchmarks have been created to test compression speeds, prime number calculation, encryption, and encoding.

Whichever benchmark you use, when comparing results between systems it's important that you understand what is really being tested. Benchmarks like those described previously often end up testing compiler optimizations between different compiler versions, rather than the benchmark code or CPU speed. Many benchmarks also execute single-threaded, but these results lose meaning in systems with multiple CPUs. A four-CPU system may benchmark slightly faster than an eight-CPU system, but the latter is likely to deliver much greater throughput when given enough parallel runnable threads.

For more on benchmarking, see Chapter 12, Benchmarking.

6.5.12 Scaling

Here is a simple scaling method, based on capacity planning of resources:

1. Determine the target user population or application request rate.
2. Express CPU usage per user or per request. For existing systems, CPU usage can be monitored with the current user count or request rate. For future systems, load generation tools can simulate users so that CPU usage can be measured.
3. Extrapolate users or requests when the CPU resources reach 100% utilization. This provides the theoretical limit for the system.

System scalability can also be modeled to account for contention and coherency latency, for a more realistic prediction of performance. See Section 2.6, Modeling,

in Chapter 2, Methodology, for more about this, and also Section 2.7, Capacity Planning, of the same chapter for more on scaling.

6.6 Analysis

This section introduces CPU performance analysis tools for Linux- and Solaris-based operating systems. See the previous section for strategies to follow when using them.

The tools in this section are listed in Table 6.6.

Table 6-6 CPU Analysis Tools

Linux	Solaris	Description
uptime	uptime	load averages
vmstat	vmstat	includes system-wide CPU averages
mpstat	mpstat	per-CPU statistics
sar	sar	historical statistics
ps	ps	process status
top	prstat	monitor per-process/thread CPU usage
pidstat	prstat	per-process/thread CPU breakdowns
time	ptime	time a command, with CPU breakdowns
DTrace, perf	DTrace	CPU profiling and tracing
perf	cpustat	CPU performance counter analysis

The list begins with tools for CPU statistics, and then drills down to tools for deeper analysis including code-path profiling and CPU cycle analysis. This is a selection of tools and capabilities to support Section 6.5, Methodology. See the documentation for each tool, including its man pages, for full references of its features.

While you may be interested in only Linux or only Solaris-based systems, consider looking at the other operating system's tools and the observability that they provide for a different perspective.

6.6.1 uptime

`uptime(1)` is one of several commands that print the system *load averages*:

```
$ uptime
  9:04pm  up 268 day(s), 10:16,  2 users,  load average: 7.76, 8.32, 8.60
```

The last three numbers are the 1-, 5-, and 15-minute load averages. By comparing the three numbers, you can determine if the load is increasing, decreasing, or steady during the last 15 minutes (or so).

Load Averages

The load average indicates the demand for CPU resources and is calculated by summing the number of threads running (utilization) and the number that are queued waiting to run (saturation). A newer method for calculating load averages uses utilization plus the sum of thread scheduler latency, rather than sampling the queue length, which improves accuracy. For reference, the internals of these calculations on Solaris-based kernels are documented in [McDougall 06b].

To interpret the value, if the load average is higher than the CPU count, there are not enough CPUs to service the threads, and some are waiting. If the load average is lower than the CPU count, it (probably) means that there is headroom, and the threads could run on-CPU when they wanted.

The three load average numbers are exponentially damped moving averages, which reflect load beyond the 1-, 5-, and 15-minute times (the times are actually constants used in the exponential moving sum [Myer 73]). Figure 6.14 shows the results of a simple experiment where a single CPU-bound thread was launched and the load averages plotted.

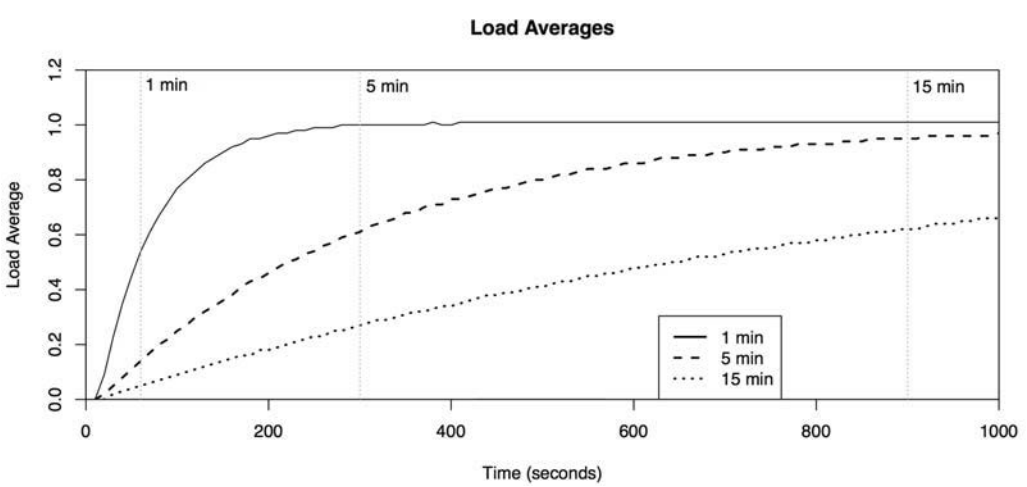


Figure 6-14 Exponentially damped load averages

By the 1-, 5-, and 15-minute marks, the load averages had reached about 61% of the known load of 1.0.

Load averages were introduced to Unix in early BSD and were based on scheduler average queue length and load averages commonly used by earlier operating systems (CTSS, Multics [Saltzer 70], TENEX [Bobrow 72]). They were described in [RFC 546]:

> [1] The TENEX load average is a measure of CPU demand. The load average is an average of the number of runable processes over a given time period. For example, an hourly load average of 10 would mean that (for a single CPU system) at any time during that hour one could expect to see 1 process running and 9 others ready to run (i.e., not blocked for I/O) waiting for the CPU.

As a modern example, a system with 64 CPUs has a load average of 128. This means that on average there is always one thread running on each CPU, and one thread waiting for each CPU. The same system with a load average of ten would indicate significant headroom, as it could run another 54 CPU-bound threads before all CPUs are busy.

Linux Load Averages

Linux currently adds tasks performing disk I/O in the uninterruptable state to the load averages. This means that the load average can no longer be interpreted to mean CPU headroom or saturation only, since it is unknown from the value alone to what degree it reflects CPU or disk load. Comparisons of the three load average numbers are also difficult, as the load may have varied among CPUs and disks over time.

A different way to incorporate other resource load is to use separate load averages for each resource type. (I've prototyped examples of this for disk, memory, and network load, each providing its own set of load averages, and found it a similar and useful overview for non-CPU resources.)

It is best to use other metrics to understand CPU load on Linux, such as those provided by `vmstat(1)` and `mpstat(1)`.

6.6.2 vmstat

The virtual memory statistics command, `vmstat(8)`, prints system-wide CPU averages in the last few columns, and a count of runnable threads in the first column. Here is example output from the **Linux** version:

```
$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
15  0   2852 46686812 279456 1401196    0    0     0     0    0    0  0  0 100  0
16  0   2852 46685192 279456 1401196    0    0     0     0 2136 36607 56 33 11  0
```

```
15  0   2852 46685952 279456 1401196    0    0     0    56 2150 36905 54 35 11  0
15  0   2852 46685960 279456 1401196    0    0     0     0 2173 36645 54 33 13  0
[...]
```

The first line of output is the summary-since-boot, with the exception of `r` on Linux—which begins by showing current values. The columns are

- **`r:`** run-queue length—the total number of runnable threads (see below)
- **`us:`** user-time
- **`sy:`** system-time (kernel)
- **`id:`** idle
- **`wa:`** wait I/O, which measures CPU idle when threads are blocked on disk I/O
- **`st:`** stolen (not shown in the output), which for virtualized environments shows CPU time spent servicing other tenants

All of these values are system-wide averages across all CPUs, with the exception of `r`, which is the total.

On **Linux**, the `r` column is the total number of tasks waiting *plus* those running. The man page currently describes it as something else—"the number of processes waiting for run time"—which suggests it counts only those waiting and not running. As insight into what this is supposed to be, the original `vmstat(1)` by Bill Joy and Ozalp Babaoglu for 3BSD in 1979 begins with an `RQ` column for the number of runnable *and* running processes, as the Linux `vmstat(8)` currently does. The man page needs updating.

On **Solaris**, the `r` column counts only the number of threads *waiting* in the dispatcher queues (run queues). The value can appear erratic, as it is sampled only once per second (from `clock()`), whereas the other CPU columns are based on high-resolution CPU microstates. These other columns currently do not include wait I/O or stolen. See Chapter 9, Disks, for more about wait I/O.

6.6.3 mpstat

The multiprocessor statistics tool, `mpstat`, can report statistics per CPU. Here is some example output from the **Linux** version:

```
$ mpstat -P ALL 1
02:47:49   CPU    %usr  %nice   %sys %iowait   %irq  %soft %steal %guest  %idle
02:47:50   all   54.37   0.00  33.12    0.00   0.00   0.00   0.00   0.00  12.50
02:47:50     0   22.00   0.00  57.00    0.00   0.00   0.00   0.00   0.00  21.00
02:47:50     1   19.00   0.00  65.00    0.00   0.00   0.00   0.00   0.00  16.00
```

continues

```
02:47:50     2   24.00   0.00  52.00    0.00   0.00   0.00   0.00   0.00  24.00
02:47:50     3  100.00   0.00   0.00    0.00   0.00   0.00   0.00   0.00   0.00
02:47:50     4  100.00   0.00   0.00    0.00   0.00   0.00   0.00   0.00   0.00
02:47:50     5  100.00   0.00   0.00    0.00   0.00   0.00   0.00   0.00   0.00
02:47:50     6  100.00   0.00   0.00    0.00   0.00   0.00   0.00   0.00   0.00
02:47:50     7   16.00   0.00  63.00    0.00   0.00   0.00   0.00   0.00  21.00
02:47:50     8  100.00   0.00   0.00    0.00   0.00   0.00   0.00   0.00   0.00
02:47:50     9   11.00   0.00  53.00    0.00   0.00   0.00   0.00   0.00  36.00
02:47:50    10  100.00   0.00   0.00    0.00   0.00   0.00   0.00   0.00   0.00
02:47:50    11   28.00   0.00  61.00    0.00   0.00   0.00   0.00   0.00  11.00
02:47:50    12   20.00   0.00  63.00    0.00   0.00   0.00   0.00   0.00  17.00
02:47:50    13   12.00   0.00  56.00    0.00   0.00   0.00   0.00   0.00  32.00
02:47:50    14   18.00   0.00  60.00    0.00   0.00   0.00   0.00   0.00  22.00
02:47:50    15  100.00   0.00   0.00    0.00   0.00   0.00   0.00   0.00   0.00
[...]
```

The `-P ALL` option was used to print the per-CPU report. By default, `mpstat(1)` prints only the system-wide summary line (`all`). The columns are

- **`CPU:`** logical CPU ID, or `all` for summary
- **`%usr:`** user-time
- **`%nice:`** user-time for processes with a nice'd priority
- **`%sys:`** system-time (kernel)
- **`%iowait:`** I/O wait
- **`%irq:`** hardware interrupt CPU usage
- **`%soft:`** software interrupt CPU usage
- **`%steal:`** time spent servicing other tenants
- **`%guest:`** CPU time spent in guest virtual machines
- **`%idle:`** idle

Key columns are `%usr`, `%sys`, and `%idle`. These identify CPU usage per CPU and show the user-time/kernel-time ratio (see Section 6.3.7, User-Time/Kernel-Time). This can also identify "hot" CPUs—those running at 100% utilization (`%usr` + `%sys`) while others are not—which can be caused by single-threaded application workloads or device interrupt mapping.

For **Solaris**-based systems, `mpstat(1M)` begins with the summary-since-boot, followed by the interval summaries. For example:

```
$ mpstat 1
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
[...]
  0 8243   0  288  3211 1265 1682   40  236  262    0  8214   47  19   0  34
  1 43708  0 1480  2753 1115 1238   58  406 1967    0 26157   17  59   0  24
  2 11987  0  393  2994 1186 1761   79  281  522    0 10035   46  21   0  34
  3 3998   0  135   935   55  238   22   60   97    0  2350   88   6   0   6
```

```
  4 12649   0  414  2885 1261 3130   82  365  619     0 14866    7  26   0  67
  5 30054   0  991   745  241 1563   52  349 1108     0 17792    8  40   0  52
  6 12882   0  439   636  167 2335   73  289  747     0 12803    6  23   0  71
  7   981   0   40   793   45  870   11   81   70     0  2022   78   3   0  19
  8  3186   0  100   687   27  450   15   75  156     0  2581   66   7   0  27
  9  8433   0  259   814  315 3382   38  280  552     0  9376    4  18   0  78
 10  8451   0  283   512  153 2158   20  194  339     0  9776    4  16   0  80
 11  3722   0  119   800  349 2693   12  199  194     0  6447    2  10   0  88
 12  4757   0  138   834  214 1387   29  142  380     0  6153   35  10   0  55
 13  5107   0  147  1404  606 3856   65  268  352     0  8188    4  14   0  82
 14  7158   0  229   672  205 1829   31  133  292     0  7637   19  12   0  69
 15  5822   0  209   866  232 1333    9  145  180     0  5164   30  13   0  57
```

The columns include

- **`CPU:`** logical CPU ID
- **`xcal:`** CPU cross calls
- **`intr:`** interrupts
- **`ithr:`** interrupts serviced as threads (lower IPL)
- **`csw:`** context switches (total)
- **`icsw:`** involuntary context switches
- **`migr:`** thread migrations
- **`smtx:`** spins on mutex locks
- **`srw:`** spins on reader/writer locks
- **`syscl:`** system calls
- **`usr:`** user-time
- **`sys:`** system-time (kernel)
- **`wt:`** wait I/O (deprecated, always zero)
- **`idl:`** idle

Key columns to check are

- **`xcal`**, to see if there is an excess rate, which consumes CPU resources. For example, look for at least 1,000/s across several CPUs. Drill-down analysis can explain their cause (see the example of this in Section 6.6.10, DTrace).
- **`smtx`**, to see if there is an excess rate, which consumes CPU resources and may also be evidence of lock contention. Lock activity can then be explored using other tools (see Chapter 5, Applications).
- **`usr`**, **`sys`**, and **`idl`**, to characterize CPU usage per CPU and the user-time/kernel-time ratio.

6.6.4 sar

The system activity reporter, `sar(1)`, can be used to observe current activity and can be configured to archive and report historical statistics. It was introduced in Chapter 4, Observability Tools, and is mentioned in other chapters as appropriate.

The **Linux** version provides the following options:

- **`-P ALL:`** same as `mpstat -P ALL`
- **`-u:`** same as `mpstat(1)`'s default output: system-wide average only
- **`-q:`** includes run-queue size as `runq-sz` (waiting plus running, the same as `vmstat`'s `r`) and load averages

The **Solaris** version provides

- **`-u:`** system-wide averages for `%usr`, `%sys`, `%wio` (zero), and `%idl`
- **`-q:`** includes run-queue size as `runq-sz` (waiting only), and percent of time the run queue had threads waiting as `%runocc`, although this value is inaccurate between 0 and 1

Per-CPU statistics are not available in the Solaris version.

6.6.5 ps

The process status command, `ps(1)`, lists details on all processes, including CPU usage statistics. For example:

```
$ ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.0  23772  1948 ?        Ss    2012   0:04 /sbin/init
root          2  0.0  0.0      0     0 ?        S     2012   0:00 [kthreadd]
root          3  0.0  0.0      0     0 ?        S     2012   0:26 [ksoftirqd/0]
root          4  0.0  0.0      0     0 ?        S     2012   0:00 [migration/0]
root          5  0.0  0.0      0     0 ?        S     2012   0:00 [watchdog/0]
[...]
web       11715 11.3  0.0 632700 11540 pts/0    Sl   01:36   0:27 node indexer.js
web       11721 96.5  0.1 638116 52108 pts/1    Rl+  01:37   3:33 node proxy.js
[...]
```

This style of operation originated from BSD and can be recognized by a lack of a dash before the `aux` options. These list all users (`a`), with extended user-oriented details (`u`), and include processes without a terminal (`x`). The terminal is shown in the teletype (`TTY`) column.

A different style, from SVR4, uses options preceded by a dash:

```
$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Nov13 ?        00:00:04 /sbin/init
root         2     0  0 Nov13 ?        00:00:00 [kthreadd]
root         3     2  0 Nov13 ?        00:00:00 [ksoftirqd/0]
root         4     2  0 Nov13 ?        00:00:00 [migration/0]
root         5     2  0 Nov13 ?        00:00:00 [watchdog/0]
[...]
```

This lists every process (`-e`) with full details (`-f`). `ps(1)` on most Linux- and Solaris-based systems supports both the BSD and SVR4 arguments.

Key columns for CPU usage are `TIME` and `%CPU`.

The `TIME` column shows the total CPU time consumed by the process (user + system) since it was created, in hours:minutes:seconds.

On **Linux**, the `%CPU` column shows the CPU usage during the previous second as the sum across all CPUs. A single-threaded CPU-bound process will report 100%. A two-thread CPU-bound process will report 200%.

On **Solaris**, `%CPU` is normalized for the CPU count. For example, a single CPU-bound thread will be shown as 12.5% for an eight-CPU system. This metric also shows *recent* CPU usage, using similar decayed averages as with load averages.

Various other options are available for `ps(1)`, including `-o` to customize the output and columns shown.

6.6.6 top

`top(1)` was created by William LeFebvre in 1984 for BSD. He was inspired by the VMS command `MONITOR PROCESS/TOPCPU`, which showed the top CPU-consuming jobs with CPU percentages and an ASCII bar chart histogram (but not columns of data).

The `top(1)` command monitors top running processes, updating the screen at regular intervals. For example, on **Linux**:

```
$ top
top - 01:38:11 up 63 days,  1:17,  2 users,  load average: 1.57, 1.81, 1.77
Tasks: 256 total,   2 running, 254 sleeping,   0 stopped,   0 zombie
Cpu(s):  2.0%us,  3.6%sy,  0.0%ni, 94.2%id,  0.0%wa,  0.0%hi,  0.2%si,  0.0%st
Mem:  49548744k total, 16746572k used, 32802172k free,   182900k buffers
Swap: 100663292k total,        0k used, 100663292k free, 14925240k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
11721 web       20   0  623m  50m 4984 R   93  0.1   0:59.50 node
11715 web       20   0  619m  20m 4916 S   25  0.0   0:07.52 node
   10 root      20   0     0    0    0 S    1  0.0 248:52.56 ksoftirqd/2
   51 root      20   0     0    0    0 S    0  0.0   0:35.66 events/0
11724 admin     20   0 19412 1444  960 R    0  0.0   0:00.07 top
    1 root      20   0 23772 1948 1296 S    0  0.0   0:04.35 init
```

A system-wide summary is at the top and a process/task listing at the bottom, sorted by the top CPU consumer by default. The system-wide summary includes the load averages and CPU states: `%us`, `%sy`, `%ni`, `%id`, `%wa`, `%hi`, `%si`, `%st`. These states are equivalent to those printed by `mpstat(1)`, as described earlier, and are averaged across all CPUs.

CPU usage is shown by the `TIME` and `%CPU` columns, which were introduced in the previous section on `ps(1)`.

This example shows a `TIME+` column, which is the same as the one shown above, but at a resolution of hundredths of a second. For example, "1:36.53" means 1 minute and 36.53 seconds of on-CPU time in total. Some versions of `top(1)` provide an optional "cumulative time" mode, which includes the CPU time from child processes that have exited.

On **Linux**, the `%CPU` column by default is not normalized by CPU count; `top(1)` calls this "Irix mode," after its behavior on IRIX. This can be switched to "Solaris mode," which divides the CPU usage by the CPU count. In that case, the hot two-thread process on a 16-CPU server would report percent CPU as 12.5.

Though `top(1)` is often a tool for beginning performance analysts, you should be aware that the CPU usage of `top(1)` itself can become significant and place `top(1)` as the top CPU-consuming process! This has been due to the available system calls—`open()`, `read()`, `close()`—and their cost when iterating over /proc entries for many processes. Some versions of `top(1)` for Solaris-based systems have reduced the overhead by leaving file descriptors open and calling `pread()`, which the `prstat(1M)` tool also does.

Since `top(1)` takes snapshots of /proc, it can miss short-lived processes that exit before a snapshot is taken. This commonly happens during software builds, where the CPUs can be heavily loaded by many short-lived tools from the build process. A variant of `top(1)` for Linux, called `atop(1)`, uses process accounting to catch the presence of short-lived processes, which it includes in its display.

6.6.7 prstat

The `prstat(1)` command was introduced as "top for Solaris-based systems." For example:

```
$ prstat
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
 21722 101        23G   20G cpu0    59    0  72:23:41 2.6% beam.smp/594
 21495 root      321M  304M sleep    1    0   2:57:41 0.9% node/5
 20721 root      345M  328M sleep    1    0   2:49:53 0.8% node/5
 20861 root      348M  331M sleep    1    0   2:57:07 0.7% node/6
 15354 root      172M  156M cpu9     1    0   0:31:42 0.7% node/5
 21738 root      179M  143M sleep    1    0   2:37:48 0.7% node/4
 20385 root      196M  174M sleep    1    0   2:26:28 0.6% node/4
```

```
  23186 root      172M  149M sleep    1    0   0:10:56 0.6% node/4
  18513 root      174M  138M cpu13    1    0   2:36:43 0.6% node/4
  21067 root      187M  162M sleep    1    0   2:28:40 0.5% node/4
  19634 root      193M  170M sleep    1    0   2:29:36 0.5% node/4
  10163 root      113M  109M sleep    1    0  12:31:09 0.4% node/3
  12699 root      199M  177M sleep    1    0   1:56:10 0.4% node/4
  37088 root     1069M 1056M sleep   59    0  38:31:19 0.3% qemu-system-x86/4
  10347 root       67M   64M sleep    1    0  11:57:17 0.3% node/3
Total: 390 processes, 1758 lwps, load averages: 3.89, 3.99, 4.31
```

A one-line system summary is at the bottom. The `CPU` column shows recent CPU usage and is the same metric shown by `top(1)` on Solaris. The `TIME` column shows consumed time.

`prstat(1M)` consumes fewer CPU resources than `top(1)` by using `pread()` to read /proc status with file descriptors left open, instead of the `open()`, `read()`, `close()` cycle.

Thread microstate accounting statistics can be printed by `prstat(1M)` using the `-m` option. The following example uses `-L` to report this per thread (per LWP) and `-c` for continual output (instead of screen refreshes):

```
$ prstat -mLc 1
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
30650 root      20 2.7 0.0 0.0 0.0 0.0  76 0.5 839  36  5K   0 node/1
42370 root      11 2.0 0.0 0.0 0.0 0.0  87 0.1 205  23  2K   0 node/1
42501 root      11 1.9 0.0 0.0 0.0 0.0  87 0.1 201  24  2K   0 node/1
42232 root      11 1.9 0.0 0.0 0.0 0.0  87 0.1 205  25  2K   0 node/1
42080 root      11 1.9 0.0 0.0 0.0 0.0  87 0.1 201  24  2K   0 node/1
53036 root     7.0 1.4 0.0 0.0 0.0 0.0  92 0.1 158  22  1K   0 node/1
56318 root     6.8 1.4 0.0 0.0 0.0 0.0  92 0.1 154  21  1K   0 node/1
55302 root     6.8 1.3 0.0 0.0 0.0 0.0  92 0.1 156  23  1K   0 node/1
54823 root     6.7 1.3 0.0 0.0 0.0 0.0  92 0.1 154  23  1K   0 node/1
54445 root     6.7 1.3 0.0 0.0 0.0 0.0  92 0.1 156  24  1K   0 node/1
53551 root     6.7 1.3 0.0 0.0 0.0 0.0  92 0.1 153  20  1K   0 node/1
21722 103      6.3 1.5 0.0 0.0 3.3 0.0  88 0.0  40   0  1K   0 beam.smp/578
21722 103      6.2 1.3 0.0 0.0 8.7 0.0  84 0.0  43   0  1K   0 beam.smp/585
21722 103      5.1 1.2 0.0 0.0 3.2 0.0  90 0.0  38   1  1K   0 beam.smp/577
21722 103      4.7 1.1 0.0 0.0 0.0 0.0  87 0.0  45   0 985   0 beam.smp/580
Total: 390 processes, 1758 lwps, load averages: 3.92, 3.99, 4.31
```

The eight highlighted columns show time spent in each microstate and sum to 100%. They are

- **`USR:`** user-time
- **`SYS:`** system-time (kernel)
- **`TRP:`** system trap
- **`TFL:`** text faults (page faults for executable segments)
- **`DFL:`** data faults

- **LCK:** time spent waiting for user-level locks
- **SLP:** time spent sleeping, including blocked on I/O
- **LAT:** scheduler latency (dispatcher queue latency)

This breakdown of thread time is extremely useful. Here are suggested paths for further investigation (also see Section 5.4.1, Thread State Analysis, in Chapter 5, Applications):

- **USR:** profiling of user-level CPU usage
- **SYS:** check system calls used and profile kernel-level CPU usage
- **SLP:** depends on the sleep event; trace syscall or code path for more details
- **LAT:** check system-wide CPU utilization and any imposed CPU limit/quota

Many of these can also be performed using DTrace.

6.6.8 pidstat

The Linux `pidstat(1)` tool prints CPU usage by process or thread, including user- and system-time breakdowns. By default, a rolling output is printed of only active processes. For example:

```
$ pidstat 1
Linux 2.6.35-32-server (dev7)   11/12/12        _x86_64_        (16 CPU)

22:24:42          PID    %usr %system  %guest    %CPU   CPU  Command
22:24:43         7814    0.00    1.98    0.00    1.98     3  tar
22:24:43         7815   97.03    2.97    0.00  100.00    11  gzip

22:24:43          PID    %usr %system  %guest    %CPU   CPU  Command
22:24:44          448    0.00    1.00    0.00    1.00     0  kjournald
22:24:44         7814    0.00    2.00    0.00    2.00     3  tar
22:24:44         7815   97.00    3.00    0.00  100.00    11  gzip
22:24:44         7816    0.00    2.00    0.00    2.00     2  pidstat
[...]
```

This example captured a system backup, involving a `tar(1)` command to read files from the file system, and the `gzip(1)` command to compress them. The user-time for `gzip(1)` is high, as expected, as it becomes CPU-bound in compression code. The `tar(1)` command spends more time in the kernel, reading from the file system.

The `-p ALL` option can be used to print all processes, including those that are idle. `-t` prints per-thread statistics. Other `pidstat(1)` options are included in other chapters of this book.

6.6.9 time, ptime

The `time(1)` command can be used to run programs and report CPU usage. It is provided either in the operating system under /usr/bin, or as a shell built-in.

This example runs time twice on a `cksum(1)` command, calculating the checksum of a large file:

```
$ time cksum Fedora-16-x86_64-Live-Desktop.iso
560560652 633339904 Fedora-16-x86_64-Live-Desktop.iso

real    0m5.105s
user    0m2.810s
sys     0m0.300s
$ time cksum Fedora-16-x86_64-Live-Desktop.iso
560560652 633339904 Fedora-16-x86_64-Live-Desktop.iso

real    0m2.474s
user    0m2.340s
sys     0m0.130s
```

The first run took 5.1 s, during which 2.8 s was in user mode—calculating the checksum—and 0.3 s was in system-time—the system calls required to read the file. There is a missing 2.0 s (5.1 - 2.8 - 0.3), which is likely time spent blocked on disk I/O reads, as this file was only partially cached. The second run completed more quickly, in 2.5 s, with almost no time blocked on I/O. This is expected, as the file may be fully cached in main memory for the second run.

On **Linux**, the `/usr/bin/time` version supports verbose details:

```
$ /usr/bin/time -v cp fileA fileB
        Command being timed: "cp fileA fileB"
        User time (seconds): 0.00
        System time (seconds): 0.26
        Percent of CPU this job got: 24%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:01.08
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 3792
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 294
        Voluntary context switches: 1082
        Involuntary context switches: 1
        Swaps: 0
        File system inputs: 275432
        File system outputs: 275432
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

The -v option is not typically provided in the shell built-in version.

Solaris-based systems include an additional `ptime(1)` version of `time(1)`, which provides high-precision times based on thread microstate accounting. Nowadays, `time(1)` on Solaris-based systems ultimately uses the same source of statistics. `ptime(1)` is still useful, as it provides a `-m` option to print the full set of thread microstate times, including scheduler latency (`lat`):

```
$ ptime -m cp fileA fileB

real        8.334800250
user        0.016714684
sys         1.899085951
trap        0.000003874
tflt        0.000000000
dflt        0.000000000
kflt        0.000000000
lock        0.000000000
slp         6.414634340
lat         0.004249234
stop        0.000285583
```

In this case, the runtime was 8.3 s, during which 6.4 s was sleeping (disk I/O).

6.6.10 DTrace

DTrace can be used to profile CPU usage for both user- and kernel-level code, and to trace the execution of functions, CPU cross calls, interrupts, and the kernel scheduler. These abilities support workload characterization, profiling, drill-down analysis, and latency analysis.

The following sections introduce DTrace for CPU analysis on Solaris- and Linux-based systems. Unless noted, the DTrace commands are intended for both operating systems. A DTrace primer was included in Chapter 4, Observability Tools.

Kernel Profiling

Previous tools, including `mpstat(1)` and `top(1)`, showed system-time—CPU time spent in the kernel. DTrace can be used to identify what the kernel is doing.

The following one-liner, demonstrated on a Solaris-based system, samples kernel stack traces at 997 Hz (to avoid lockstep, as explained in Section 6.5.4, Profiling). The predicate ensures that the CPU is in kernel mode when sampling, by checking that the kernel program counter (`arg0`) is nonzero:

```
# dtrace -n 'profile-997 /arg0/ { @[stack()] = count(); }'
dtrace: description 'profile-997 ' matched 1 probe
^C
```

```
[...]
              unix`do_copy_fault_nta+0x49
              genunix`uiomove+0x12e
              zfs`dmu_write_uio_dnode+0xac
              zfs`dmu_write_uio_dbuf+0x54
              zfs`zfs_write+0xc60
              genunix`fop_write+0x8b
              genunix`write+0x250
              genunix`write32+0x1e
              unix`_sys_sysenter_post_swapgs+0x149
              302

              unix`do_splx+0x65
              genunix`disp_lock_exit+0x47
              genunix`post_syscall+0x318
              genunix`syscall_exit+0x68
              unix`0xffffffffb800ed9
              621

              unix`i86_mwait+0xd
              unix`cpu_idle_mwait+0x109
              unix`idle+0xa7
              unix`thread_start+0x8
            23083
```

The most frequent stack is printed last, which in this case is for the idle thread, which was sampled 23,083 times. For the other stacks, the top function and ancestry are shown.

Many pages were truncated from this output. The following one-liners show other ways to sample kernel CPU usage, some of which condense the output much further.

One-Liners

Sample kernel stacks at 997 Hz:

```
dtrace -n 'profile-997 /arg0/ { @[stack()] = count(); }'
```

Sample kernel stacks at 997 Hz, top ten only:

```
dtrace -n 'profile-997 /arg0/ { @[stack()] = count(); } END { trunc(@, 10); }'
```

Sample kernel stacks, five frames only, at 997 Hz:

```
dtrace -n 'profile-997 /arg0/ { @[stack(5)] = count(); }'
```

Sample kernel on-CPU functions at 997 Hz:

```
dtrace -n 'profile-997 /arg0/ { @[func(arg0)] = count(); }'
```

Sample kernel on-CPU modules at 997 Hz:

```
dtrace -n 'profile-997 /arg0/ { @[mod(arg0)] = count(); }'
```

User Profiling

CPU time spent in user mode can be profiled similarly to the kernel. The following one-liner matches on user-level code by checking on `arg1` (user PC) and also matches processes named `"mysqld"` (MySQL database):

```
# dtrace -n 'profile-97 /arg1 && execname == "mysqld"/ { @[ustack()] =
    count(); }'
dtrace: description 'profile-97 ' matched 1 probe
^C
[...]
libc.so.1`__priocntlset+0xa
libc.so.1`getparam+0x83
libc.so.1`pthread_getschedparam+0x3c
libc.so.1`pthread_setschedprio+0x1f
mysqld`_Z16dispatch_command19enum_server_commandP3THDPcj+0x9ab
mysqld`_Z10do_commandP3THD+0x198
mysqld`handle_one_connection+0x1a6
libc.so.1`_thrp_setup+0x8d
libc.so.1`_lwp_start
4884

mysqld`_Z13add_to_statusP17system_status_varS0_+0x47
mysqld`_Z22calc_sum_of_all_statusP17system_status_var+0x67
mysqld`_Z16dispatch_command19enum_server_commandP3THDPcj+0x1222
mysqld`_Z10do_commandP3THD+0x198
mysqld`handle_one_connection+0x1a6
libc.so.1`_thrp_setup+0x8d
libc.so.1`_lwp_start
5530
```

The last stack shows that MySQL was in `do_command()` and performing `calc_sum_of_all_status()`, which was frequently on-CPU. The stack frames look a little mangled as they are C++ signatures (the `c++filt(1)` tool can be used to unmangle them).

The following one-liners show other ways to sample user CPU usage, provided user-level actions are available (this feature is currently not yet ported to Linux).

One-Liners

Sample user stacks at 97 Hz, for PID 123:

```
dtrace -n 'profile-97 /arg1 && pid == 123/ { @[ustack()] = count(); }'
```

Sample user stacks at 97 Hz, for all processes named "sshd":

```
dtrace -n 'profile-97 /arg1 && execname == "sshd"/ { @[ustack()] = count(); }'
```

Sample user stacks at 97 Hz, for all processes on the system (include process name in output):

```
dtrace -n 'profile-97 /arg1/ { @[execname, ustack()] = count(); }'
```

Sample user stacks at 97 Hz, top ten only, for PID 123:

```
dtrace -n 'profile-97 /arg1 && pid == 123/ { @[ustack()] = count(); }
    END { trunc(@, 10); }'
```

Sample user stacks, five frames only, at 97 Hz, for PID 123:

```
dtrace -n 'profile-97 /arg1 && pid == 123/ { @[ustack(5)] = count(); }'
```

Sample user on-CPU functions at 97 Hz, for PID 123:

```
dtrace -n 'profile-97 /arg1 && pid == 123/ { @[ufunc(arg1)] = count(); }'
```

Sample user on-CPU modules at 97 Hz, for PID 123:

```
dtrace -n 'profile-97 /arg1 && pid == 123/ { @[umod(arg1)] = count(); }'
```

Sample user stacks at 97 Hz, including during system-time when the user stack is frozen (typically on a syscall), for PID 123:

```
dtrace -n 'profile-97 /pid == 123/ { @[ustack()] = count(); }'
```

Sample which CPU a process runs on, at 97 Hz, for PID 123:

```
dtrace -n 'profile-97 /pid == 123/ { @[cpu] = count(); }'
```

Function Tracing

While profiling can show the total CPU time consumed by functions, it doesn't show the runtime distribution of those function calls. This can be determined by using tracing and the `vtimestamp` built-in—a high-resolution timestamp that increments only when the current thread is on-CPU. A function's CPU time can be measured by tracing its entry and return and calculating the `vtimestamp` delta.

For example, using dynamic tracing (fbt provider) to measure the CPU time in the kernel ZFS `zio_checksum_generate()` function:

```
# dtrace -n 'fbt::zio_checksum_generate:entry { self->v = vtimestamp; }
    fbt::zio_checksum_generate:return /self->v/ { @["ns"] =
    quantize(vtimestamp - self->v); self->v = 0; }'
dtrace: description 'fbt::zio_checksum_generate:entry ' matched 2 probes
^C

  ns
           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |                                         3
             512 |@                                        62
            1024 |@                                        79
            2048 |                                         13
            4096 |                                         21
            8192 |                                         8
           16384 |                                         2
           32768 |                                         41
           65536 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     3740
          131072 |@                                        134
          262144 |                                         0
```

Most of the time this function took between 65 and 131 µs of CPU time. This includes the CPU time of all subfunctions.

This particular style of tracing can add overhead if the function is called frequently. It is best used in conjunction with profiling, so that results can be cross-checked.

Similar dynamic tracing may be performed for user-level code via the PID provider, if available.

Dynamic tracing via the fbt or pid providers is considered an unstable interface, as functions may change between releases. There are static tracing providers available for tracing CPU behavior, which are intended to provide a stable interface. These include probes for CPU cross calls, interrupts, and scheduler activity.

CPU Cross Calls

Excessive CPU cross calls can reduce performance due to their CPU consumption. Prior to DTrace, the origin of cross calls was difficult to determine. It's now as easy as a one-liner, tracing cross calls and showing the code path that led to them:

```
# dtrace -n 'sysinfo:::xcalls { @[stack()] = count(); }'
dtrace: description 'sysinfo:::xcalls ' matched 1 probe
^C
[...]
              unix`xc_sync+0x39
              kvm`kvm_xcall+0xa9
              kvm`vcpu_clear+0x1d
              kvm`vmx_vcpu_load+0x3f
              kvm`kvm_arch_vcpu_load+0x16
              kvm`kvm_ctx_restore+0x3d
              genunix`restorectx+0x37
              unix`_resume_from_idle+0x83
               97
```

This was demonstrated on a Solaris-based system with the sysinfo provider.

Interrupts

DTrace allows interrupts to be traced and examined. **Solaris**-based systems ship with `intrstat(1M)`, a DTrace-based tool for summarizing interrupt CPU usage. For example:

```
# intrstat 1
[...]
      device |      cpu4 %tim      cpu5 %tim      cpu6 %tim      cpu7 %tim
-------------+------------------------------------------------------------
       bnx#0 |         0  0.0         0  0.0         0  0.0         0  0.0
      ehci#0 |         0  0.0         0  0.0         0  0.0         0  0.0
      ehci#1 |         0  0.0         0  0.0         0  0.0         0  0.0
       igb#0 |         0  0.0         0  0.0         0  0.0         0  0.0
  mega_sas#0 |         0  0.0      5585  7.1         0  0.0         0  0.0
      uhci#0 |         0  0.0         0  0.0         0  0.0         0  0.0
      uhci#1 |         0  0.0         0  0.0         0  0.0         0  0.0
      uhci#2 |         0  0.0         0  0.0         0  0.0         0  0.0
      uhci#3 |         0  0.0         0  0.0         0  0.0         0  0.0
[...]
```

The output is typically pages long on multi-CPU systems and includes interrupt counts and percent CPU times for each driver, for each CPU. The preceding excerpt shows that the `mega_sas` driver was consuming 7.1% of CPU 5.

If `intrstat(1M)` is not available (as is currently the case on **Linux**), interrupt activity can be examined by use of dynamic function tracing.

Scheduler Tracing

The scheduler provider (sched) provides probes for tracing operations of the kernel CPU scheduler. Probes are listed in Table 6.7.

Table 6-7 sched Provider Probes

Probe	Description
on-cpu	The current thread begins execution on-CPU.
off-cpu	The current thread is about to end execution on-CPU.
remain-cpu	The scheduler has decided to continue running the current thread.
enqueue	A thread is being enqueued to a run queue (examine it via `args[]`).
dequeue	A thread is being dequeued from a run queue (examine it via `args[]`).
preempt	The current thread is about to be preempted by another.

Since many of these fire in thread context, the `curthread` built-in refers to the thread in question, and thread-local variables can be used. For example, tracing on-CPU runtime using a thread-local variable (`self->ts`):

```
# dtrace -n 'sched:::on-cpu /execname == "sshd"/ { self->ts = timestamp; }
    sched:::off-cpu /self->ts/ { @["ns"] = quantize(timestamp - self->ts);
    self->ts = 0; }'
dtrace: description 'sched:::on-cpu ' matched 6 probes
^C

  ns
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |                                         1
            8192 |@@                                       8
           16384 |@@@                                      12
           32768 |@@@@@@@@@@@@@@@@@@@@@@@@                 94
           65536 |@@@                                      14
          131072 |@@@                                      12
          262144 |@@                                       7
          524288 |@                                        4
         1048576 |@                                        5
         2097152 |                                         2
         4194304 |                                         1
         8388608 |                                         1
        16777216 |                                         0
```

This traced the on-CPU runtime for processes named "sshd". Most of the time it was on-CPU only briefly, between 32 and 65 µs.

6.6.11 SystemTap

SystemTap can also be used on Linux systems for tracing of scheduler events. See Section 4.4, SystemTap, in Chapter 4, Observability Tools, and Appendix E for help with converting the previous DTrace scripts.

6.6.12 perf

Originally called Performance Counters for Linux (PCL), the `perf(1)` command has evolved and become a collection of tools for profiling and tracing, now called Linux Performance Events (LPE). Each tool is selected as a subcommand. For example, `perf stat` executes the stat command, which provides CPC-based statistics. These commands are listed in the USAGE message, and a selection is reproduced here in Table 6.8 (from version 3.2.6-3).

Table 6-8 perf Subcommands

Command	Description
`annotate`	Read perf.data (created by perf record) and display annotated code.
`diff`	Read two perf.data files and display the differential profile.
`evlist`	List the event names in a perf.data file.
`inject`	Filter to augment the events stream with additional information.
`kmem`	Tool to trace/measure kernel memory (slab) properties.
`kvm`	Tool to trace/measure kvm guest OS.
`list`	List all symbolic event types.
`lock`	Analyze lock events.
`probe`	Define new dynamic tracepoints.
`record`	Run a command and record its profile into perf.data.
`report`	Read perf.data (created by perf record) and display the profile.
`sched`	Tool to trace/measure scheduler properties (latencies).
`script`	Read perf.data (created by perf record) and display trace output.
`stat`	Run a command and gather performance counter statistics.
`timechart`	Tool to visualize total system behavior during a workload.
`top`	System profiling tool.

Key commands are demonstrated in the following sections.

System Profiling

`perf(1)` can be used to profile CPU call paths, summarizing where CPU time is spent in both kernel- and user-space. This is performed by the `record` command, which captures samples at regular intervals to a perf.data file. A `report` command is then used to view the file.

In the following example, all CPUs (`-a`) are sampled with call stacks (`-g`) at 997 Hz (`-F 997`) for 10 s (`sleep 10`). The `--stdio` option is used to print all the output, instead of operating in interactive mode.

```
# perf record -a -g -F 997 sleep 10
[ perf record: Woken up 44 times to write data ]
[ perf record: Captured and wrote 13.251 MB perf.data (~578952 samples) ]
# perf report --stdio
[...]
# Overhead      Command      Shared Object                                 Symbol
# ........  ...........  ..................  ..............................
#
    72.98%      swapper  [kernel.kallsyms]  [k] native_safe_halt
                |
                --- native_safe_halt
                    default_idle
                    cpu_idle
                    rest_init
                    start_kernel
                    x86_64_start_reservations
                    x86_64_start_kernel

     9.43%           dd  [kernel.kallsyms]  [k] acpi_pm_read
                     |
                     --- acpi_pm_read
                         ktime_get_ts
                         |
                         |--87.75%-- __delayacct_blkio_start
                         |          io_schedule_timeout
                         |          balance_dirty_pages_ratelimited_nr
                         |          generic_file_buffered_write
                         |          __generic_file_aio_write
                         |          generic_file_aio_write
                         |          ext4_file_write
                         |          do_sync_write
                         |          vfs_write
                         |          sys_write
                         |          system_call
                         |          __GI___libc_write
                         |
[...]
```

The full output is many pages long, in descending sample count order. These sample counts are given as percentages, which show where the CPU time was spent. This example indicates that 72.98% of time was spent in the idle thread, and 9.43% of time in the `dd` process. Out of that 9.43%, 87.5% is composed of the stack shown, which is for `ext4_file_write()`.

These kernel and process symbols are available only if their debuginfo files are available; otherwise hex addresses are shown.

`perf(1)` operates by programming an overflow interrupt for the CPU cycle counter. Since the cycle rate varies on modern processors, a "scaled" counter is used that remains constant.

Process Profiling

Apart from profiling across all CPUs, individual processes can be targeted. The following command executes the *command* and creates the perf.data file:

```
# perf record -g command
```

As before, debuginfo must be available for `perf(1)` to translate symbols when viewing the report.

Scheduler Latency

The `sched` command records and reports scheduler statistics. For example:

```
# perf sched record sleep 10
[ perf record: Woken up 108 times to write data ]
[ perf record: Captured and wrote 1723.874 MB perf.data (~75317184 samples) ]
# perf sched latency

 -----------------------------------------------------------------------------------------
---------------------------
  Task                  |   Runtime ms  | Switches | Average delay ms | Maximum delay
ms | Maximum delay at     |
 -----------------------------------------------------------------------------------------
---------------------------
  kblockd/0:91          |      0.009 ms |        1 | avg:    1.193 ms | max:    1.193
ms | max at: 105455.615096 s
  dd:8439               |   9691.404 ms |      763 | avg:    0.363 ms | max:   29.953
ms | max at: 105456.540771 s
  perf_2.6.35-32:8440   |   8082.543 ms |      818 | avg:    0.362 ms | max:   29.956
ms | max at: 105460.734775 s
  kjournald:419         |    462.561 ms |      457 | avg:    0.064 ms | max:   12.112
ms | max at: 105459.815203 s
[...]
  INFO: 0.976% lost events (167317 out of 17138781, in 3 chunks)
  INFO: 0.178% state machine bugs (4766 out of 2673759) (due to lost events?)
  INFO: 0.000% context switch bugs (3 out of 2673759) (due to lost events?)
```

This shows the average and maximum scheduler latency while tracing.

Scheduler events are frequent, so this type of tracing incurs CPU and storage overhead. The perf.data file in this example was 1.7 Gbytes for 10 s of tracing. The `INFO` lines in the output show that some events were dropped. This points out an advantage of the DTrace model of in-kernel filtering and aggregation: it can

summarize data while tracing and pass only the summary to user-space, minimizing overhead.

stat

The `stat` command provides a high-level summary of CPU cycle behavior based on CPC. In the following example it launches a `gzip(1)` command:

```
$ perf stat gzip file1

 Performance counter stats for 'gzip perf.data':

       62250.620881  task-clock-msecs         #      0.998 CPUs
                 65  context-switches         #      0.000 M/sec
                  1  CPU-migrations           #      0.000 M/sec
                211  page-faults              #      0.000 M/sec
       149282502161  cycles                   #   2398.089 M/sec
       227631116972  instructions             #      1.525 IPC
        39078733567  branches                 #    627.765 M/sec
         1802924170  branch-misses            #      4.614 %
           87791362  cache-references         #      1.410 M/sec
           24187334  cache-misses             #      0.389 M/sec

       62.355529199  seconds time elapsed
```

The statistics include the cycle and instruction count, and the IPC (inverse of CPI). As described earlier, this is an extremely useful high-level metric for determining the types of cycles occurring and how many of them are stall cycles.

The following lists other counters that can be examined:

```
# perf list

List of pre-defined events (to be used in -e):

  cpu-cycles OR cycles                      [Hardware event]
  instructions                              [Hardware event]
  cache-references                          [Hardware event]
  cache-misses                              [Hardware event]
  branch-instructions OR branches           [Hardware event]
  branch-misses                             [Hardware event]
  bus-cycles                                [Hardware event]
[...]
  L1-dcache-loads                           [Hardware cache event]
  L1-dcache-load-misses                     [Hardware cache event]
  L1-dcache-stores                          [Hardware cache event]
  L1-dcache-store-misses                    [Hardware cache event]
[...]
```

Look for both “Hardware event” and “Hardware cache event.” Those available depend on the processor architecture and are documented in the processor manuals (e.g., the Intel *Software Developer’s Manual*).

These events can be specified using –e. For example (this is from an Intel Xeon):

```
$ perf stat -e instructions,cycles,L1-dcache-load-misses,LLC-load-misses,dTLB-load-
misses gzip file1

 Performance counter stats for 'gzip file1':

       12278136571  instructions              #      2.199 IPC
        5582247352  cycles
          90367344  L1-dcache-load-misses
           1227085  LLC-load-misses
            685149  dTLB-load-misses

       2.332492555  seconds time elapsed
```

Apart from instructions and cycles, this example also measured the following:

- **L1-dcache-load-misses:** Level 1 data cache load misses. This gives you a measure of the memory load caused by the application, after some loads have been returned from the Level 1 cache. It can be compared with other L1 event counters to determine cache hit rate.
- **LLC-load-misses:** Last level cache load misses. After the last level, this accesses main memory, and so this is a measure of main memory load. The difference between this and `L1-dcache-load-misses` gives an idea (other counters are needed for completeness) of the effectiveness of the CPU caches beyond Level 1.
- **dTLB-load-misses:** Data translation lookaside buffer misses. This shows the effectiveness of the MMU to cache page mappings for the workload and can measure the size of the memory workload (working set).

Many other counters can be inspected. `perf(1)` supports both descriptive names (like those used for this example) and hexadecimal values. The latter may be necessary for esoteric counters you find in the processor manuals, for which a descriptive name isn't provided.

Software Tracing

`perf record -e` can be used with various software instrumentation points for tracing activity of the kernel scheduler. These include software events and tracepoint events (static probes), as listed by `perf list`. For example:

```
# perf list
  context-switches OR cs                             [Software event]
  cpu-migrations OR migrations                       [Software event]
[...]
```

continues

```
  sched:sched_kthread_stop                           [Tracepoint event]
  sched:sched_kthread_stop_ret                       [Tracepoint event]
  sched:sched_wakeup                                 [Tracepoint event]
  sched:sched_wakeup_new                             [Tracepoint event]
  sched:sched_switch                                 [Tracepoint event]
  sched:sched_migrate_task                           [Tracepoint event]
  sched:sched_process_free                           [Tracepoint event]
  sched:sched_process_exit                           [Tracepoint event]
  sched:sched_wait_task                              [Tracepoint event]
  sched:sched_process_wait                           [Tracepoint event]
  sched:sched_process_fork                           [Tracepoint event]
  sched:sched_stat_wait                              [Tracepoint event]
  sched:sched_stat_sleep                             [Tracepoint event]
  sched:sched_stat_iowait                            [Tracepoint event]
  sched:sched_stat_runtime                           [Tracepoint event]
  sched:sched_pi_setprio                             [Tracepoint event]
[...]
```

The following example uses the context switch software event to trace when applications leave the CPU and collects call stacks for 10 s:

```
# perf record -f -g -a -e context-switches sleep 10
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.417 MB perf.data (~18202 samples) ]
# perf report --stdio
# ========
# captured on: Wed Apr 10 19:52:19 2013
# hostname : 9d219ce8-cf52-409f-a14a-b210850f3231
[...]
#
# Events: 2K context-switches
#
# Overhead    Command      Shared Object      Symbol
# ........  .........  ..................  ..........
#
    47.60%       perl  [kernel.kallsyms]  [k] __schedule
                 |
                 --- __schedule
                     schedule
                     retint_careful
                    |
                    |--50.11%-- Perl_pp_unstack
                    |
                    |--26.40%-- Perl_pp_stub
                    |
                     --23.50%-- Perl_runops_standard

    25.66%        tar  [kernel.kallsyms]  [k] __schedule
                  |
                  --- __schedule
                      |
                      |--99.72%-- schedule
                      |           |
                      |           |--99.90%-- io_schedule
                      |           |           sleep_on_buffer
                      |           |           __wait_on_bit
                      |           |           out_of_line_wait_on_bit
                      |           |           __wait_on_buffer
                      |           |           |
                      |           |           |--99.21%-- ext4_bread
                      |           |           |          |
```

```
                    |          |          |          |--99.72%-- htree_dirbl...
                    |          |          |          |           ext4_htree_f...
                    |          |          |          |           ext4_readdir
                    |          |          |          |           vfs_readdir
                    |          |          |          |           sys_getdents
                    |          |          |          |           system_call
                    |          |          |          |           __getdents64
                    |          |          |          |--0.28%-- [...]
                    |          |          |
                    |          |          |--0.79%-- __ext4_get_inode_loc
[...]
```

This truncated output shows two applications, perl and tar, and their call stacks when they context switched. Reading the stacks shows the tar program was sleeping on file system (ext4) reads. The perl program was involuntary context switched as it is performing heavy compute, although that isn't clear from this output alone.

More information can be found using the sched tracepoint events. Kernel scheduler functions can also be traced directly using *dynamic tracepoints* (dynamic tracing), which along with the static probes can provide similar data to what was seen earlier from DTrace, although it can require more post-processing to produce the results you are after.

Chapter 9, Disks, includes another example of static tracing with `perf(1)`: block I/O tracepoints. Chapter 10, Network, includes an example of dynamic tracing with `perf(1)` for the `tcp_sendmsg()` kernel function.

Documentation

For more on `perf(1)`, see its man pages, documentation in the Linux kernel source under tools/perf/Documentation, the "Perf Tutorial" [4], and "The Unofficial Linux Perf Events Web-Page" [5].

6.6.13 cpustat

On **Solaris**-based systems, the tools for examining CPC are `cpustat(1M)` for system-wide analysis and `cputrack(1M)` for process analysis. These refer to CPC using the term *performance instrumentation counters* (PICs).

For example, to measure CPI, both cycles and instructions must be counted. Using the PAPI names:

```
# cpustat -tc PAPI_tot_cyc,PAPI_tot_ins,sys 1
   time cpu event       tsc      pic0      pic1
  1.001   0  tick 2390794244 2095800691 910588497
  1.002   1  tick 2391617432 2091867238 832659178
  1.002   2  tick 2392676108 2075492108 917078382
  1.003   3  tick 2393561424 2067362862 831551337
  1.003   4  tick 2393739432 2020553426 909065542
[...]
```

`cpustat(1M)` produces a line of output per CPU. This output can be post-processed (e.g., with awk) so that the CPI calculation can be made.

The `sys` token was used so that both user- and kernel-mode cycles are counted. This sets the flag described in CPU Performance Counters in Section 6.4.1, Hardware.

Measuring the same counters using the platform-specific event names:

```
# cpustat -tc cpu_clk_unhalted.thread_p,inst_retired.any_p,sys 1
```

Run `cpustat -h` for the full list of supported counters for your processor. The output usually ends with a reference to the vendor processor manual; for example:

> See Appendix A of the "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2" Order Number: 253669-026US, February 2008.

The manuals describe low-level processor behavior in detail.

Only one instance of `cpustat(1M)` can be running on the system at the same time, as the kernel does not support multiplexing.

6.6.14 Other Tools

Other **Linux** CPU performance tools include

- **oprofile:** the original CPU profiling tool by John Levon.
- **htop:** includes ASCII bar charts for CPU usage and has a more powerful interactive interface than the original `top(1)`.
- **atop:** includes many more system-wide statistics and uses process accounting to catch the presence of short-lived processes.
- **/proc/cpuinfo:** This can be read to see processor details, including clock speed and feature flags.
- **getdelays.c:** This is an example of delay accounting observability and includes CPU scheduler latency per process. It was demonstrated in Chapter 4, Observability Tools.
- **valgrind:** a memory debugging and profiling toolkit [6]. It contains callgrind, a tool to trace function calls and gather a call graph, which can be visualized using kcachegrind; and cachegrind for analysis of hardware cache usage by a given program.

For **Solaris**:

- **`lockstat/plockstat:`** for lock analysis, including spin locks and CPU consumption from adaptive mutexes (see Chapter 5, Applications).
- **`psrinfo:`** processor status and information (`-vp`).
- **`fmadm faulty:`** to check if a CPU has been predictively faulted due to an increase in correctable ECC errors. Also see `fmstat(1M)`.
- **`isainfo -x:`** to list processor feature flags.
- **`pginfo`**, **`pgstat:`** processor group statistics, showing CPU topology and how CPU resources are shared.
- **`lgrpinfo:`** for locality group statistics. This can be useful for checking that lgrps are in use, which requires processor and operating system support.

There are also sophisticated products for CPU performance analysis, including Oracle Solaris Studio, which is available for Solaris and Linux.

6.6.15 Visualizations

CPU usage has historically been visualized as line graphs of utilization or load average, including the original X11 load tool (`xload(1)`). Such line graphs are an effective way to show variation, as magnitudes can be visually compared. They can also show patterns over time, as was shown in Section 2.9, Monitoring, of Chapter 2, Methodology.

However, line graphs of per-CPU utilization don't scale with the CPU counts we see today, especially for cloud computing environments involving tens of thousands of CPUs—a graph of 10,000 lines can become paint.

Other statistics plotted as line graphs, including averages, standard deviations, maximums, and percentiles, provide some value and do scale. However, CPU utilization is often *bimodal*—composed of idle or near-idle CPUs, and then some at 100% utilization—which is not effectively conveyed with these statistics. The full distribution often needs to be studied. A utilization heat map makes this possible.

The following sections introduce CPU utilization heat maps, CPU subsecond-offset heat maps, and flame graphs. I created these visualization types to solve problems in enterprise and cloud performance analysis.

Utilization Heat Map

Utilization versus time can be presented as a heat map, with the saturation (darkness) of each pixel showing the number of CPUs at that utilization and time range. Heat maps were introduced in Chapter 2, Methodology.

Figure 6.15 shows CPU utilization for an entire data center (availability zone), running a public cloud environment. It includes over 300 physical servers and 5,312 CPUs.

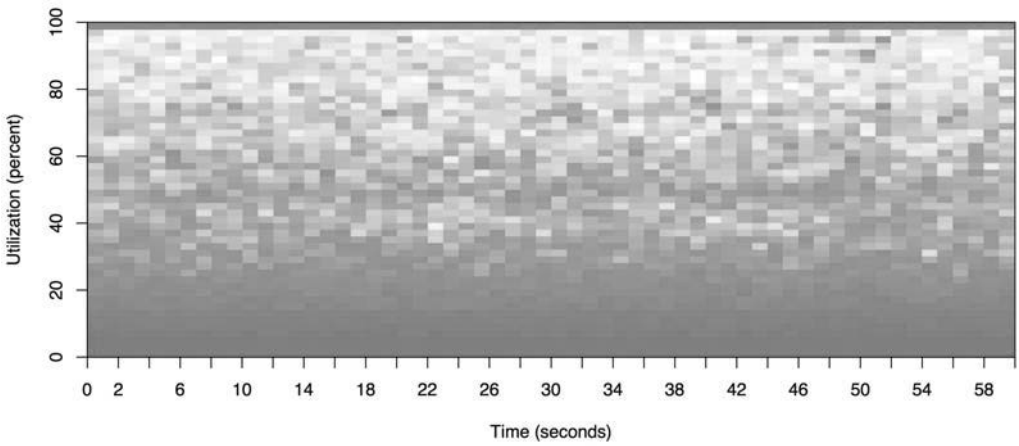


Figure 6-15 CPU utilization heat map, 5,312 CPUs

The darker shading at the bottom of this heat map shows that most CPUs are running between 0% and 30% utilization. However, the solid line at the top shows that, over time, there are also some CPUs at 100% utilization. The fact that the line is dark shows that multiple CPUs were at 100%, not just one.

This particular visualization is provided by real-time monitoring software (Joyent Cloud Analytics), which allows points to be selected with a click to reveal more details. In this case, the 100% CPU line can be clicked to reveal which servers these CPUs belonged to, and what tenants and applications are driving CPUs at that rate.

Subsecond-Offset Heat Map

This heat map type allows activity within a second to be examined. CPU activity is typically measured in microseconds or milliseconds; reporting this data as averages over an entire second can wipe out useful information. This type of heat map puts the subsecond offset on the *y* axis, with the number of non-idle CPUs at each offset shown by the saturation. This visualizes each second as a column, "painting" it from bottom to top.

Figure 6.16 shows a CPU subsecond-offset heat map for a cloud database (Riak).

What is interesting about this heat map isn't the times that the CPUs were busy servicing the database, but the times that they were not, indicated by the

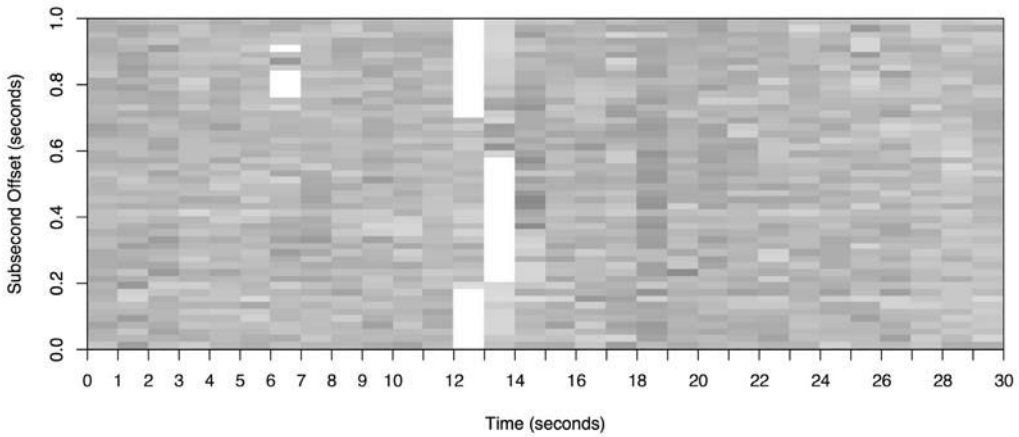


Figure 6-16 CPU subsecond-offset heat map

white columns. The duration of these gaps was also interesting: hundreds of milliseconds during which none of the database threads were on-CPU. This led to the discovery of a locking issue where the entire database was blocked for hundreds of milliseconds at a time.

If we had examined this data using a line graph, a dip in per-second CPU utilization might have been dismissed as variable load and not investigated further.

Flame Graphs

Profiling stack traces is an effective way to explain CPU usage, showing which kernel- or user-level code paths are responsible. It can, however, produce thousands of pages of output. Flame graphs visualize the profile stack frames, so that CPU usage can be understood more quickly and more clearly.

Flame graphs can be built upon data from DTrace, perf, or SystemTap. The example in Figure 6.17 shows the Linux kernel profiled using perf.

The flame graph has the following characteristics:

- Each box represents a function in the stack (a "stack frame").
- The *y* axis shows stack depth (number of frames on the stack). The top box shows the function that was on-CPU. Everything beneath that is ancestry. The function beneath a function is its parent, just as in the stack traces shown earlier.
- The *x* axis spans the sample population. It does not show the passing of time from left to right, as most graphs do. The left-to-right ordering has no meaning (it's sorted alphabetically).

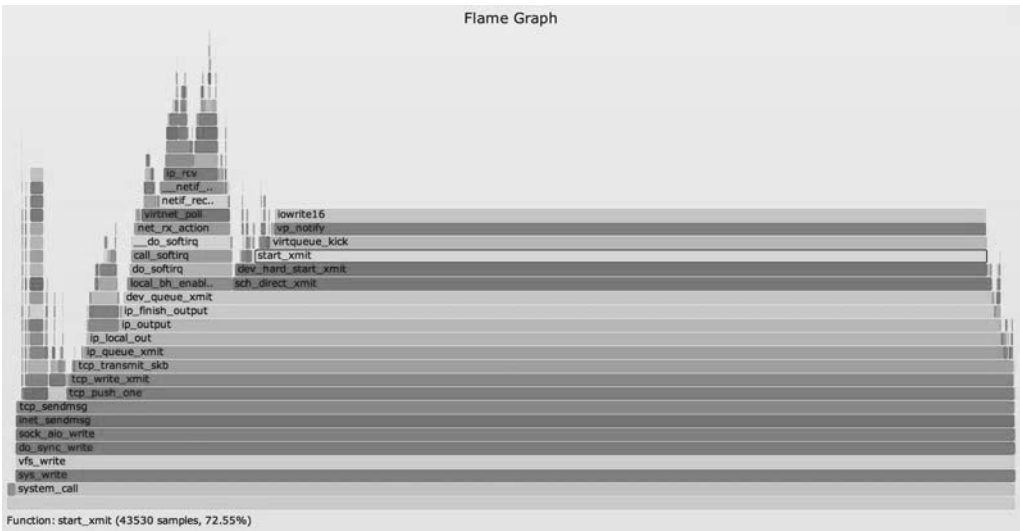


Figure 6-17 Linux kernel flame graph

- The width of the box shows the total time it was on-CPU or part of an ancestry that was on-CPU (based on sample count). Wider box functions may be slower than narrow box functions, or they may simply be called more often. The call count is not shown (nor is it known via sampling).
- The sample count can exceed elapsed time if multiple threads were running and sampled in parallel.

The colors are not significant and are picked at random to be warm colors. It's called a "flame graph" because it shows what is hot on-CPU.

It is also *interactive*. It is an SVG with an embedded JavaScript routine that when opened in a browser allows you to mouse over elements to reveal details at the bottom. In the Figure 6.17 example, `start_xmit()` was highlighted, which shows that it was present in 72.55% of the sampled stacks.

6.7 Experimentation

This section describes tools for actively testing CPU performance. See Section 6.5.11, Micro-Benchmarking, for background.

When using these tools, it's a good idea to leave `mpstat(1)` continually running to confirm CPU usage and parallelism.

6.7.1 Ad Hoc

While this is trivial and doesn't measure anything, it can be a useful known workload for confirming that observability tools show what they claim to show. This creates a single-threaded workload that is CPU-bound ("hot on one CPU"):

```
# while :; do :; done &
```

This is a Bourne shell program that performs an infinite loop in the background. It will need to be killed once you no longer need it.

6.7.2 SysBench

The SysBench system benchmark suite has a simple CPU benchmark tool that calculates prime numbers. For example:

```
# sysbench --num-threads=8 --test=cpu --cpu-max-prime=100000 run
sysbench 0.4.12:  multi-threaded system evaluation benchmark

Running the test with following options:
Number of threads: 8

Doing CPU performance benchmark

Threads started!
Done.

Maximum prime number checked in CPU test: 100000


Test execution summary:
    total time:                          30.4125s
    total number of events:              10000
    total time taken by event execution: 243.2310
    per-request statistics:
         min:                                  24.31ms
         avg:                                  24.32ms
         max:                                  32.44ms
         approx.  95 percentile:               24.32ms

Threads fairness:
    events (avg/stddev):           1250.0000/1.22
    execution time (avg/stddev):   30.4039/0.01
```

This executed eight threads, with a maximum prime number of 100,000. The runtime was 30.4 s, which can be used for comparison with the results from other systems or configurations (assuming many things, such as that identical compiler options were used to build the software; see Chapter 12, Benchmarking).

6.8 Tuning

For CPUs, the biggest performance wins are typically those that eliminate unnecessary work, which is an effective form of tuning. Section 6.5, Methodology, and Section 6.6, Analysis, introduced many ways to analyze and identify the work performed, helping you find any unnecessary work. Other methodologies for tuning were also introduced: priority tuning and CPU binding. This section includes these and other tuning examples.

The specifics of tuning—the options available and what to set them to—depend on the processor type, the operating system version, and the intended workload. The following, organized by type, provide examples of what options may be available and how they are tuned. The earlier methodology sections provide guidance on when and why these tunables would be tuned.

6.8.1 Compiler Options

Compilers, and the options they provide for code optimization, can have a dramatic effect on CPU performance. Common options include compiling for 64-bit instead of 32-bit, and selecting a level of optimizations. Compiler optimization is discussed in Chapter 5, Applications.

6.8.2 Scheduling Priority and Class

The `nice(1)` command can be used to adjust process priority. Positive nice values decrease priority, and negative nice values *increase* priority, which only the superuser can set. The range is from -20 to +19. For example:

```
$ nice -n 19 command
```

runs the command with a nice value of 19—the lowest priority that nice can set. To change the priority of an already running process, use `renice(1)`.

On **Linux**, the `chrt(1)` command can show and set the scheduling priority directly, and the scheduling policy. The scheduling priority can also be set directly using the `setpriority()` syscall, and the priority and scheduling policy can be set using the `sched_setscheduler()` syscall.

On **Solaris**, you can set scheduling classes and priorities directly using the `priocntl(1)` command. For example:

```
# priocntl -s -c RT -p 10 -i pid PID
```

This sets the target process ID to run in the real-time scheduling class with a priority of 10. Be careful when setting this: you can lock up your system if the real-time threads consume all CPU resources.

6.8.3 Scheduler Options

Your kernel may provide tunable parameters to control scheduler behavior, although it is unlikely that these will ever need to be tuned.

On **Linux** systems, config options can be set, including the examples in Table 6.9 from a 3.2.6 kernel, with defaults from Fedora 16.

Table 6-9 Example Linux Scheduler Config Options

Option	Default	Description
CONFIG_CGROUP_SCHED	y	allows tasks to be grouped, allocating CPU time on a group basis
CONFIG_FAIR_GROUP_SCHED	y	allows CFS tasks to be grouped
CONFIG_RT_GROUP_SCHED	y	allows real-time tasks to be grouped
CONFIG_SCHED_AUTOGROUP	y	automatically identifies and creates task groups (e.g., build jobs)
CONFIG_SCHED_SMT	y	hyperthreading support
CONFIG_SCHED_MC	y	multicore support
CONFIG_HZ	1,000	sets kernel clock rate (timer interrupt)
CONFIG_NO_HZ	y	tickless kernel behavior
CONFIG_SCHED_HRTICK	y	use high-resolution timers
CONFIG_PREEMPT	n	full kernel preemption (exception of spin lock regions and interrupts)
CONFIG_PREEMPT_NONE	n	no preemption
CONFIG_PREEMPT_VOLUNTARY	y	preemption at voluntary kernel code points

Some Linux kernels provide additional tunables (e.g., in /proc/sys/sched).

On **Solaris**-based systems, the kernel tunable parameters shown in Table 6.10 modify scheduler behavior.

For reference, find the matching documentation for your operating system version (e.g., for Solaris, the *Solaris Tunable Parameters Reference Manual*). Such documentation should list key tunable parameters, their type, when to set them, their defaults, and the valid ranges. Be careful when using these, as their ranges may not be fully tested. (Tuning them may also be prohibited by company or vendor policy.)

Table 6-10 Example Solaris Scheduler Tunables

Parameter	Default	Description
rechoose_interval	3	CPU affinity duration (clock ticks)
nosteal_nsec	100,000	avoid thread steals (idle CPU looking for work) if thread ran this recently (nanoseconds)
hires_tick	0	change to 1 for a 1,000 Hz kernel clock rate, instead of 100 Hz

Scheduler Class Tuning

Solaris-based systems also provide a means to modify the time quantum and priorities used by scheduling classes, via the `dispadmin(1)` command. For example, printing out the table of tunables (called the *dispatcher table*) for the time-sharing scheduling class (TS):

```
# dispadmin -c TS -g -r 1000
# Time Sharing Dispatcher Configuration
RES=1000

# ts_quantum  ts_tqexp  ts_slpret  ts_maxwait ts_lwait  PRIORITY LEVEL
        200         0         50           0        50        #     0
        200         0         50           0        50        #     1
        200         0         50           0        50        #     2
        200         0         50           0        50        #     3
        200         0         50           0        50        #     4
        200         0         50           0        50        #     5
[...]
```

This output includes

- **`ts_quantum:`** time quantum (in milliseconds, as set resolution using `-r 1000`)
- **`ts_tqexp:`** new priority provided when the thread expires its current time quantum (priority reduction)
- **`ts_slpret:`** new priority after thread sleeps (I/O) then wakes up (priority promotion)
- **`ts_maxwait:`** maximum seconds waiting for CPU before being promoted to the priority in `ts_lwait`
- **`PRIORITY LEVEL:`** priority value

This can be written to a file, modified, then reloaded by `dispadmin(1M)`. You ought to have a reason for doing this, such as having first measured priority contention and scheduler latency using DTrace.

6.8.4 Process Binding

A process may be bound to one or more CPUs, which may increase its performance by improving cache warmth and memory locality.

On **Linux**, this is performed using the `taskset(1)` command, which can use a CPU mask or ranges to set CPU affinity. For example:

```
$ taskset -pc 7-10 10790
pid 10790's current affinity list: 0-15
pid 10790's new affinity list: 7-10
```

This sets PID 10790 to run only on CPUs 7 through 10.

On **Solaris**-based systems, this is performed using `pbind(1)`. For example:

```
$ pbind -b 10 11901
process id 11901: was not bound, now 10
```

This sets PID 11901 to run on CPU 10. Multiple CPUs cannot be specified. For similar functionality, use exclusive CPU sets.

6.8.5 Exclusive CPU Sets

Linux provides *cpusets*, which allow CPUs to be grouped and processes assigned to them. This can improve performance similarly to process binding, but performance can be improved further by making the cpuset exclusive—preventing other processes from using it. The trade-off is a reduction in available CPU for the rest of the system.

The following commented example creates an exclusive set:

```
# mkdir /dev/cpuset
# mount -t cpuset cpuset /dev/cpuset
# cd /dev/cpuset
# mkdir prodset             # create a cpuset called "prodset"
# cd prodset
# echo 7-10 > cpus          # assign CPUs 7-10
# echo 1 > cpu_exclusive    # make prodset exclusive
# echo 1159 > tasks         # assign PID 1159 to prodset
```

For reference, see the `cpuset(7)` man page.

On **Solaris**, you can create exclusive CPU sets using the `psrset(1M)` command.

6.8.6 Resource Controls

Apart from associating processes with whole CPUs, modern operating systems provide resource controls for fine-grained allocation of CPU usage.

Solaris-based systems have resource controls (added in Solaris 9) for processes or groups of processes called *projects*. CPU usage can be controlled in a flexible way using the fair share scheduler and *shares*, which control how idle CPU can be consumed by those who need it. Limits can also be imposed, in terms of total percent CPU utilization, for cases where consistency is more desirable than the dynamic behavior of shares.

For **Linux**, there are container groups (cgroups), which can also control resource usage by processes or groups of processes. CPU usage can be controlled using shares, and the CFS scheduler allows fixed limits to be imposed (*CPU bandwidth*), in terms of allocating microseconds of CPU cycles per interval. CPU bandwidth is relatively new, added in 2012 (3.2).

Chapter 11, Cloud Computing, describes a use case of managing CPU usage of OS-virtualized tenants, including how shares and limits can be used in concert.

6.8.7 Processor Options (BIOS Tuning)

Processors typically provide settings to enable, disable, and tune processor-level features. On x86 systems, these are typically accessed via the BIOS settings menu at boot time.

The settings usually provide maximum performance by default and don't need to be adjusted. The most common reason I adjust these today is to disable Intel Turbo Boost, so that CPU benchmarks execute with a consistent clock rate (bearing in mind that, for production use, Turbo Boost should be enabled for slightly faster performance).

6.9 Exercises

1. Answer the following questions about CPU terminology:
   - What is the difference between a process and a processor?
   - What is a hardware thread?
   - What is the run queue (also called a dispatcher queue)?
   - What is the difference between user-time and kernel-time?
   - What is CPI?

2. Answer the following conceptual questions:
   - Describe CPU utilization and saturation.
   - Describe how the instruction pipeline improves CPU throughput.
   - Describe how processor instruction width improves CPU throughput.
   - Describe the advantages of multiprocess and multithreaded models.
3. Answer the following deeper questions:
   - Describe what happens when the system CPUs are overloaded with runnable work, including the effect on application performance.
   - When there is no runnable work to perform, what do the CPUs do?
   - When handed a suspected CPU performance issue, name three methodologies you would use early during the investigation, and explain why.
4. Develop the following procedures for your operating system:
   - A USE method checklist for CPU resources. Include how to fetch each metric (e.g., which command to execute) and how to interpret the result. Try to use existing OS observability tools before installing or using additional software products.
   - A workload characterization checklist for CPU resources. Include how to fetch each metric, and try to use existing OS observability tools first.
5. Perform these tasks:
   - Calculate the load average for the following system, whose load is at steady state:
     - The system has 64 CPUs.
     - The system-wide CPU utilization is 50%.
     - The system-wide CPU saturation, measured as the total number of runnable and queued threads on average, is 2.0.
   - Choose an application, and profile its user-level CPU usage. Show which code paths are consuming the most CPU.
   - Describe CPU behavior visible from this Solaris-based screen shot alone:

```
# prstat -mLc 10
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
 11076 mysql    4.3 0.7 0.0 0.0 0.0  58  31 5.7 790  48 12K   0 mysqld/15620
 11076 mysql    3.5 1.0 0.0 0.0 0.0  42  46 7.6  1K  42 18K   0 mysqld/15189
 11076 mysql    3.0 0.9 0.0 0.0 0.0  34  53 8.9  1K  20 17K   0 mysqld/14454
 11076 mysql    3.1 0.6 0.0 0.0 0.0  55  36 5.7 729  27 11K   0 mysqld/15849
 11076 mysql    2.5 1.1 0.0 0.0 0.0  28  59 8.6  1K  35 19K   0 mysqld/16094
 11076 mysql    2.4 1.1 0.0 0.0 0.0  34  54 8.3  1K  45 20K   0 mysqld/16304
```

continues

```
  11076 mysql    2.5 0.8 0.0 0.0 0.0  56  32 8.8  1K  16 15K   0 mysqld/16181
  11076 mysql    2.3 1.1 0.0 0.0 0.0 8.5  79 9.0  1K  21 20K   0 mysqld/15856
  11076 mysql    2.3 1.0 0.0 0.0 0.0  12  76 9.2  1K  40 16K   0 mysqld/15411
  11076 mysql    2.2 1.0 0.0 0.0 0.0  29  57  11  1K  53 17K   0 mysqld/16277
  11076 mysql    2.2 0.8 0.0 0.0 0.0  36  54 7.1 993  27 15K   0 mysqld/16266
  11076 mysql    2.1 0.8 0.0 0.0 0.0  34  56 7.1  1K  19 16K   0 mysqld/16320
  11076 mysql    2.3 0.7 0.0 0.0 0.0  44  47 5.8 831  24 12K   0 mysqld/15971
  11076 mysql    2.1 0.7 0.0 0.0 0.0  54  37 5.3 862  22 13K   0 mysqld/15442
  11076 mysql    1.9 0.9 0.0 0.0 0.0  45  46 6.3  1K  23 16K   0 mysqld/16201
Total: 34 processes, 333 lwps, load averages: 32.68, 35.47, 36.12
```

6. (optional, advanced) Develop `bustop(1)`—a tool that shows physical bus or interconnect utilization—with a presentation similar to `iostat(1)`: a list of busses, columns for throughput in each direction, and utilization. Include saturation and error metrics if possible. This will require using CPC.

6.10 References

[Saltzer 70] Saltzer, J., and J. Gintell. "The Instrumentation of Multics," *Communications of the ACM*, August 1970.

[Bobrow 72] Bobrow, D., et al. "TENEX: A Paged Time Sharing System for the PDP-10*," *Communications of the ACM*, March 1972.

[Myer 73] Myer, T. H., J. R. Barnaby, and W. W. Plummer. *TENEX Executive Manual*. Bolt, Baranek and Newman, Inc., April 1973.

[Hinnant 84] Hinnant, D. "Benchmarking UNIX Systems," *BYTE* magazine 9, no. 8 (August 1984).

[Bulpin 05] Bulpin, J., and I. Pratt. "Hyper-Threading Aware Process Scheduling Heuristics," USENIX, 2005.

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

[Otto 06] Otto, E. *Temperature-Aware Operating System Scheduling* (Thesis). University of Virginia, 2006.

[Ruggiero 08] Ruggiero, J. *Measuring Cache and Memory Latency and CPU to Memory Bandwidth.* Intel (Whitepaper), 2008.

[Intel 09] *An Introduction to the Intel QuickPath Interconnect*. Intel, 2009.

[Intel 12] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Combined Volumes 1, 2A, 2B, 2C, 3A, 3B, and 3C. Intel, 2012.

[Intel 13] *Intel 64 and IA-32 Architectures Software Developer's Manual,* Volume 3B, *System Programming Guide, Part 2.* Intel, 2013.

[RFC 546] *TENEX Load Averages for July 1973,* August 1973. http://tools.ietf.org/html/rfc546.

[1] http://lwn.net/Articles/178253/

[2] www.bitmover.com/lmbench/

[3] http://minnie.tuhs.org/cgi-bin/utree.pl?file=V4

[4] https://perf.wiki.kernel.org/index.php/Tutorial

[5] www.eece.maine.edu/~vweaver/projects/perf_events

[6] http://valgrind.org/docs/manual/

This page intentionally left blank

This page intentionally left blank

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z