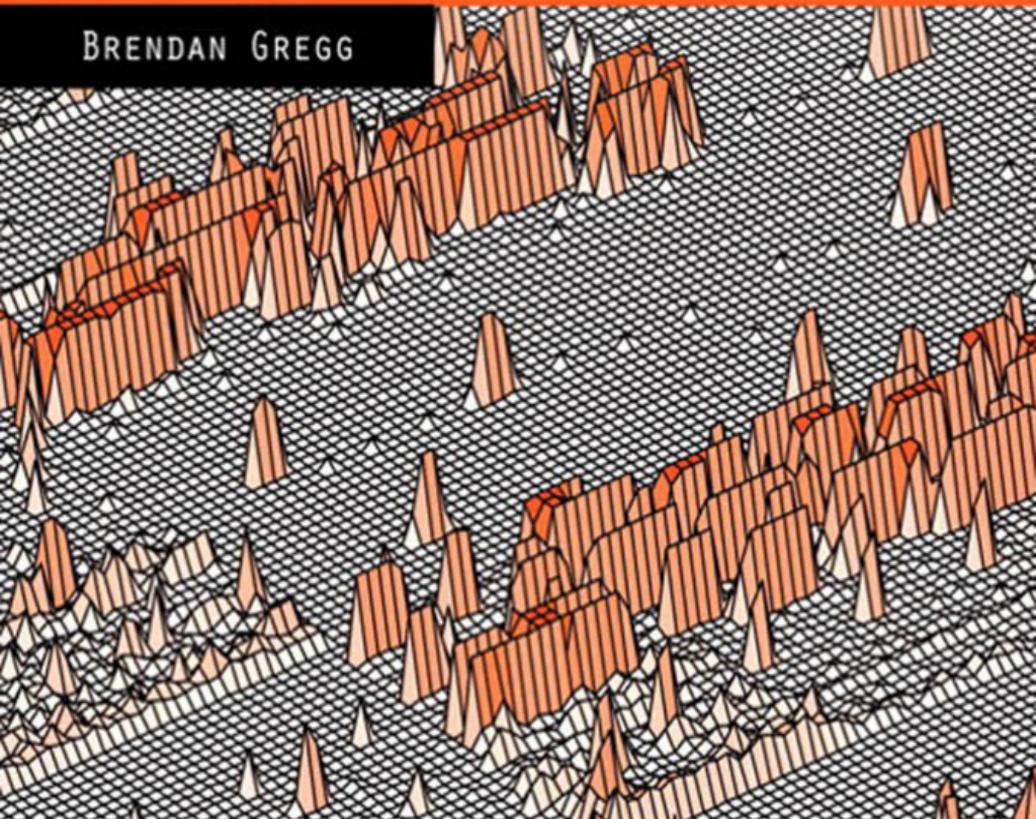


Systems Performance

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



PRENTICE
HALL

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	xxv
Acknowledgments	xxxiii
About the Author	xxxv
Chapter 1 Introduction	1
1.1 Systems Performance	1
1.2 Roles	2
1.3 Activities	3
1.4 Perspectives	4
1.5 Performance Is Challenging	4
1.5.1 Performance Is Subjective	5
1.5.2 Systems Are Complex	5
1.5.3 There Can Be Multiple Performance Issues	6
1.6 Latency	6
1.7 Dynamic Tracing	7
1.8 Cloud Computing	8
1.9 Case Studies	9
1.9.1 Slow Disks	9

1.9.2	Software Change	11
1.9.3	More Reading	13
Chapter 2	Methodology	15
2.1	Terminology	16
2.2	Models	17
2.2.1	System under Test	17
2.2.2	Queueing System	17
2.3	Concepts	18
2.3.1	Latency	18
2.3.2	Time Scales	19
2.3.3	Trade-offs	20
2.3.4	Tuning Efforts	21
2.3.5	Level of Appropriateness	22
2.3.6	Point-in-Time Recommendations	23
2.3.7	Load versus Architecture	24
2.3.8	Scalability	24
2.3.9	Known-Unknowns	26
2.3.10	Metrics	27
2.3.11	Utilization	27
2.3.12	Saturation	29
2.3.13	Profiling	30
2.3.14	Caching	30
2.4	Perspectives	32
2.4.1	Resource Analysis	33
2.4.2	Workload Analysis	34
2.5	Methodology	35
2.5.1	Streetlight Anti-Method	36
2.5.2	Random Change Anti-Method	37
2.5.3	Blame-Someone-Else Anti-Method	38
2.5.4	Ad Hoc Checklist Method	38
2.5.5	Problem Statement	39
2.5.6	Scientific Method	39

2.5.7	Diagnosis Cycle	41
2.5.8	Tools Method	41
2.5.9	The USE Method	42
2.5.10	Workload Characterization	49
2.5.11	Drill-Down Analysis	50
2.5.12	Latency Analysis	51
2.5.13	Method R	52
2.5.14	Event Tracing	53
2.5.15	Baseline Statistics	54
2.5.16	Static Performance Tuning	55
2.5.17	Cache Tuning	55
2.5.18	Micro-Benchmarking	56
2.6	Modeling	57
2.6.1	Enterprise versus Cloud	57
2.6.2	Visual Identification	58
2.6.3	Amdahl's Law of Scalability	60
2.6.4	Universal Scalability Law	61
2.6.5	Queueing Theory	61
2.7	Capacity Planning	65
2.7.1	Resource Limits	66
2.7.2	Factor Analysis	68
2.7.3	Scaling Solutions	69
2.8	Statistics	69
2.8.1	Quantifying Performance	69
2.8.2	Averages	70
2.8.3	Standard Deviations, Percentiles, Median	72
2.8.4	Coefficient of Variation	72
2.8.5	Multimodal Distributions	73
2.8.6	Outliers	74
2.9	Monitoring	74
2.9.1	Time-Based Patterns	74
2.9.2	Monitoring Products	76
2.9.3	Summary-since-Boot	76

2.10	Visualizations	76
2.10.1	Line Chart	77
2.10.2	Scatter Plots	78
2.10.3	Heat Maps	79
2.10.4	Surface Plot	80
2.10.5	Visualization Tools	81
2.11	Exercises	82
2.12	References	82
Chapter 3	Operating Systems	85
3.1	Terminology	86
3.2	Background	87
3.2.1	Kernel	87
3.2.2	Stacks	89
3.2.3	Interrupts and Interrupt Threads	91
3.2.4	Interrupt Priority Level	92
3.2.5	Processes	93
3.2.6	System Calls	95
3.2.7	Virtual Memory	97
3.2.8	Memory Management	97
3.2.9	Schedulers	98
3.2.10	File Systems	99
3.2.11	Caching	101
3.2.12	Networking	102
3.2.13	Device Drivers	103
3.2.14	Multiprocessor	103
3.2.15	Preemption	103
3.2.16	Resource Management	104
3.2.17	Observability	104
3.3	Kernels	105
3.3.1	Unix	106
3.3.2	Solaris-Based	106
3.3.3	Linux-Based	109
3.3.4	Differences	112

3.4 Exercises	113
3.5 References	113
Chapter 4 Observability Tools	115
4.1 Tool Types	116
4.1.1 Counters	116
4.1.2 Tracing	118
4.1.3 Profiling	119
4.1.4 Monitoring (sar)	120
4.2 Observability Sources	120
4.2.1 /proc	121
4.2.2 /sys	126
4.2.3 kstat	127
4.2.4 Delay Accounting	130
4.2.5 Microstate Accounting	131
4.2.6 Other Observability Sources	131
4.3 DTrace	133
4.3.1 Static and Dynamic Tracing	134
4.3.2 Probes	135
4.3.3 Providers	136
4.3.4 Arguments	137
4.3.5 D Language	137
4.3.6 Built-in Variables	137
4.3.7 Actions	138
4.3.8 Variable Types	139
4.3.9 One-Liners	141
4.3.10 Scripting	141
4.3.11 Overheads	143
4.3.12 Documentation and Resources	143
4.4 SystemTap	144
4.4.1 Probes	145
4.4.2 Tapsets	145
4.4.3 Actions and Built-ins	146

4.4.4	Examples	146
4.4.5	Overheads	148
4.4.6	Documentation and Resources	149
4.5	perf	149
4.6	Observing Observability	150
4.7	Exercises	151
4.8	References	151
Chapter 5	Applications	153
5.1	Application Basics	153
5.1.1	Objectives	155
5.1.2	Optimize the Common Case	156
5.1.3	Observability	156
5.1.4	Big O Notation	156
5.2	Application Performance Techniques	158
5.2.1	Selecting an I/O Size	158
5.2.2	Caching	158
5.2.3	Buffering	159
5.2.4	Polling	159
5.2.5	Concurrency and Parallelism	160
5.2.6	Non-Blocking I/O	162
5.2.7	Processor Binding	163
5.3	Programming Languages	163
5.3.1	Compiled Languages	164
5.3.2	Interpreted Languages	165
5.3.3	Virtual Machines	166
5.3.4	Garbage Collection	166
5.4	Methodology and Analysis	167
5.4.1	Thread State Analysis	168
5.4.2	CPU Profiling	171
5.4.3	Syscall Analysis	173
5.4.4	I/O Profiling	180
5.4.5	Workload Characterization	181

5.4.6 USE Method	181
5.4.7 Drill-Down Analysis	182
5.4.8 Lock Analysis	182
5.4.9 Static Performance Tuning	185
5.5 Exercises	186
5.6 References	187
Chapter 6 CPUs	189
6.1 Terminology	190
6.2 Models	191
6.2.1 CPU Architecture	191
6.2.2 CPU Memory Caches	191
6.2.3 CPU Run Queues	192
6.3 Concepts	193
6.3.1 Clock Rate	193
6.3.2 Instruction	193
6.3.3 Instruction Pipeline	194
6.3.4 Instruction Width	194
6.3.5 CPI, IPC	194
6.3.6 Utilization	195
6.3.7 User-Time/Kernel-Time	196
6.3.8 Saturation	196
6.3.9 Preemption	196
6.3.10 Priority Inversion	196
6.3.11 Multiprocess, Multithreading	197
6.3.12 Word Size	198
6.3.13 Compiler Optimization	199
6.4 Architecture	199
6.4.1 Hardware	199
6.4.2 Software	209
6.5 Methodology	214
6.5.1 Tools Method	215
6.5.2 USE Method	216

6.5.3	Workload Characterization	216
6.5.4	Profiling	218
6.5.5	Cycle Analysis	219
6.5.6	Performance Monitoring	220
6.5.7	Static Performance Tuning	220
6.5.8	Priority Tuning	221
6.5.9	Resource Controls	222
6.5.10	CPU Binding	222
6.5.11	Micro-Benchmarking	222
6.5.12	Scaling	223
6.6	Analysis	224
6.6.1	uptime	224
6.6.2	vmstat	226
6.6.3	mpstat	227
6.6.4	sar	230
6.6.5	ps	230
6.6.6	top	231
6.6.7	prstat	232
6.6.8	pidstat	234
6.6.9	time, ptime	235
6.6.10	DTrace	236
6.6.11	SystemTap	243
6.6.12	perf	243
6.6.13	cpustat	249
6.6.14	Other Tools	250
6.6.15	Visualizations	251
6.7	Experimentation	254
6.7.1	Ad Hoc	255
6.7.2	SysBench	255
6.8	Tuning	256
6.8.1	Compiler Options	256
6.8.2	Scheduling Priority and Class	256
6.8.3	Scheduler Options	257

6.8.4	Process Binding	259
6.8.5	Exclusive CPU Sets	259
6.8.6	Resource Controls	260
6.8.7	Processor Options (BIOS Tuning)	260
6.9	Exercises	260
6.10	References	262
Chapter 7	Memory	265
7.1	Terminology	266
7.2	Concepts	267
7.2.1	Virtual Memory	267
7.2.2	Paging	268
7.2.3	Demand Paging	269
7.2.4	Overcommit	270
7.2.5	Swapping	271
7.2.6	File System Cache Usage	271
7.2.7	Utilization and Saturation	271
7.2.8	Allocators	272
7.2.9	Word Size	272
7.3	Architecture	272
7.3.1	Hardware	273
7.3.2	Software	278
7.3.3	Process Address Space	284
7.4	Methodology	289
7.4.1	Tools Method	289
7.4.2	USE Method	290
7.4.3	Characterizing Usage	291
7.4.4	Cycle Analysis	293
7.4.5	Performance Monitoring	293
7.4.6	Leak Detection	293
7.4.7	Static Performance Tuning	294
7.4.8	Resource Controls	294
7.4.9	Micro-Benchmarking	294

7.5	Analysis	295
7.5.1	vmstat	295
7.5.2	sar	298
7.5.3	slabtop	302
7.5.4	::kmastat	302
7.5.5	ps	304
7.5.6	top	305
7.5.7	prstat	305
7.5.8	pmap	306
7.5.9	DTrace	308
7.5.10	SystemTap	312
7.5.11	Other Tools	312
7.6	Tuning	314
7.6.1	Tunable Parameters	314
7.6.2	Multiple Page Sizes	317
7.6.3	Allocators	318
7.6.4	Resource Controls	318
7.7	Exercises	319
7.8	References	320
Chapter 8	File Systems	323
8.1	Terminology	324
8.2	Models	325
8.2.1	File System Interfaces	325
8.2.2	File System Cache	325
8.2.3	Second-Level Cache	326
8.3	Concepts	326
8.3.1	File System Latency	327
8.3.2	Caching	327
8.3.3	Random versus Sequential I/O	328
8.3.4	Prefetch	329
8.3.5	Read-Ahead	330
8.3.6	Write-Back Caching	330

8.3.7 Synchronous Writes	331
8.3.8 Raw and Direct I/O	331
8.3.9 Non-Blocking I/O	332
8.3.10 Memory-Mapped Files	332
8.3.11 Metadata	333
8.3.12 Logical versus Physical I/O	333
8.3.13 Operations Are Not Equal	335
8.3.14 Special File Systems	336
8.3.15 Access Timestamps	336
8.3.16 Capacity	337
8.4 Architecture	337
8.4.1 File System I/O Stack	337
8.4.2 VFS	337
8.4.3 File System Caches	339
8.4.4 File System Features	344
8.4.5 File System Types	345
8.4.6 Volumes and Pools	351
8.5 Methodology	353
8.5.1 Disk Analysis	353
8.5.2 Latency Analysis	354
8.5.3 Workload Characterization	356
8.5.4 Performance Monitoring	358
8.5.5 Event Tracing	358
8.5.6 Static Performance Tuning	359
8.5.7 Cache Tuning	360
8.5.8 Workload Separation	360
8.5.9 Memory-Based File Systems	360
8.5.10 Micro-Benchmarking	361
8.6 Analysis	362
8.6.1 vfsstat	363
8.6.2 fsstat	364
8.6.3 strace, truss	364
8.6.4 DTrace	365

8.6.5	SystemTap	375
8.6.6	LatencyTOP	375
8.6.7	free	376
8.6.8	top	376
8.6.9	vmstat	376
8.6.10	sar	377
8.6.11	slabtop	378
8.6.12	mdb ::kmastat	379
8.6.13	fcachestat	379
8.6.14	/proc/meminfo	380
8.6.15	mdb ::memstat	380
8.6.16	kstat	381
8.6.17	Other Tools	382
8.6.18	Visualizations	383
8.7	Experimentation	383
8.7.1	Ad Hoc	384
8.7.2	Micro-Benchmark Tools	384
8.7.3	Cache Flushing	387
8.8	Tuning	387
8.8.1	Application Calls	387
8.8.2	ext3	389
8.8.3	ZFS	389
8.9	Exercises	391
8.10	References	392
Chapter 9	Disks	395
9.1	Terminology	396
9.2	Models	397
9.2.1	Simple Disk	397
9.2.2	Caching Disk	397
9.2.3	Controller	398
9.3	Concepts	399
9.3.1	Measuring Time	399

9.3.2	Time Scales	400
9.3.3	Caching	401
9.3.4	Random versus Sequential I/O	402
9.3.5	Read/Write Ratio	403
9.3.6	I/O Size	403
9.3.7	IOPS Are Not Equal	404
9.3.8	Non-Data-Transfer Disk Commands	404
9.3.9	Utilization	404
9.3.10	Saturation	405
9.3.11	I/O Wait	406
9.3.12	Synchronous versus Asynchronous	407
9.3.13	Disk versus Application I/O	407
9.4	Architecture	407
9.4.1	Disk Types	408
9.4.2	Interfaces	414
9.4.3	Storage Types	415
9.4.4	Operating System Disk I/O Stack	418
9.5	Methodology	421
9.5.1	Tools Method	422
9.5.2	USE Method	422
9.5.3	Performance Monitoring	423
9.5.4	Workload Characterization	424
9.5.5	Latency Analysis	426
9.5.6	Event Tracing	427
9.5.7	Static Performance Tuning	428
9.5.8	Cache Tuning	429
9.5.9	Resource Controls	429
9.5.10	Micro-Benchmarking	429
9.5.11	Scaling	431
9.6	Analysis	431
9.6.1	iostat	432
9.6.2	sar	440
9.6.3	pidstat	441

9.6.4 DTrace	442
9.6.5 SystemTap	451
9.6.6 perf	451
9.6.7 iotop	452
9.6.8 iosnoop	455
9.6.9 blktrace	457
9.6.10 MegaCli	459
9.6.11 smartctl	460
9.6.12 Visualizations	461
9.7 Experimentation	465
9.7.1 Ad Hoc	465
9.7.2 Custom Load Generators	465
9.7.3 Micro-Benchmark Tools	466
9.7.4 Random Read Example	466
9.8 Tuning	467
9.8.1 Operating System Tunables	467
9.8.2 Disk Device Tunables	469
9.8.3 Disk Controller Tunables	469
9.9 Exercises	470
9.10 References	471
Chapter 10 Network	473
10.1 Terminology	474
10.2 Models	474
10.2.1 Network Interface	474
10.2.2 Controller	475
10.2.3 Protocol Stack	476
10.3 Concepts	476
10.3.1 Networks and Routing	476
10.3.2 Protocols	477
10.3.3 Encapsulation	478
10.3.4 Packet Size	478
10.3.5 Latency	479

10.3.6	Buffering	481
10.3.7	Connection Backlog	481
10.3.8	Interface Negotiation	482
10.3.9	Utilization	482
10.3.10	Local Connections	482
10.4	Architecture	483
10.4.1	Protocols	483
10.4.2	Hardware	486
10.4.3	Software	488
10.5	Methodology	493
10.5.1	Tools Method	494
10.5.2	USE Method	495
10.5.3	Workload Characterization	496
10.5.4	Latency Analysis	497
10.5.5	Performance Monitoring	498
10.5.6	Packet Sniffing	498
10.5.7	TCP Analysis	500
10.5.8	Drill-Down Analysis	500
10.5.9	Static Performance Tuning	501
10.5.10	Resource Controls	502
10.5.11	Micro-Benchmarking	502
10.6	Analysis	503
10.6.1	netstat	503
10.6.2	sar	509
10.6.3	ifconfig	511
10.6.4	ip	512
10.6.5	nicstat	512
10.6.6	dladm	513
10.6.7	ping	514
10.6.8	traceroute	514
10.6.9	pathchar	515
10.6.10	tcpdump	516
10.6.11	snoop	517

10.6.12	Wireshark	520
10.6.13	DTrace	520
10.6.14	SystemTap	533
10.6.15	perf	533
10.6.16	Other Tools	534
10.7	Experimentation	535
10.7.1	iperf	535
10.8	Tuning	536
10.8.1	Linux	536
10.8.2	Solaris	539
10.8.3	Configuration	542
10.9	Exercises	542
10.10	References	543
Chapter 11	Cloud Computing	545
11.1	Background	546
11.1.1	Price/Performance Ratio	546
11.1.2	Scalable Architecture	547
11.1.3	Capacity Planning	548
11.1.4	Storage	550
11.1.5	Multitenancy	550
11.2	OS Virtualization	551
11.2.1	Overhead	553
11.2.2	Resource Controls	555
11.2.3	Observability	558
11.3	Hardware Virtualization	563
11.3.1	Overhead	566
11.3.2	Resource Controls	572
11.3.3	Observability	574
11.4	Comparisons	581
11.5	Exercises	583
11.6	References	584

Chapter 12	Benchmarking	587
12.1	Background	588
12.1.1	Activities	588
12.1.2	Effective Benchmarking	589
12.1.3	Benchmarking Sins	591
12.2	Benchmarking Types	597
12.2.1	Micro-Benchmarking	597
12.2.2	Simulation	599
12.2.3	Replay	600
12.2.4	Industry Standards	601
12.3	Methodology	602
12.3.1	Passive Benchmarking	603
12.3.2	Active Benchmarking	604
12.3.3	CPU Profiling	606
12.3.4	USE Method	607
12.3.5	Workload Characterization	608
12.3.6	Custom Benchmarks	608
12.3.7	Ramping Load	608
12.3.8	Sanity Check	611
12.3.9	Statistical Analysis	612
12.4	Benchmark Questions	613
12.5	Exercises	614
12.6	References	615
Chapter 13	Case Study	617
13.1	Case Study: The Red Whale	617
13.1.1	Problem Statement	618
13.1.2	Support	619
13.1.3	Getting Started	620
13.1.4	Choose Your Own Adventure	622
13.1.5	The USE Method	623
13.1.6	Are We Done?	626
13.1.7	Take 2	627

13.1.8	The Basics	628
13.1.9	Ignoring the Red Whale	628
13.1.10	Interrogating the Kernel	629
13.1.11	Why?	631
13.1.12	Epilogue	633
13.2	Comments	633
13.3	Additional Information	634
13.4	References	634
Appendix A	USE Method: Linux	637
	Physical Resources	637
	Software Resources	640
	Reference	641
Appendix B	USE Method: Solaris	643
	Physical Resources	643
	Software Resources	646
	References	647
Appendix C	sar Summary	649
	Linux	649
	Solaris	650
Appendix D	DTrace One-Liners	651
	syscall Provider	651
	proc Provider	655
	profile Provider	655
	sched Provider	657
	fbt Provider	658
	pid Provider	659
	io Provider	660
	sysinfo Provider	660
	vminfo Provider	661
	ip Provider	661

tcp provider	662
udp provider	663
Appendix E DTrace to SystemTap	665
Functionality	665
Terminology	666
Probes	666
Built-in Variables	667
Functions	668
Example 1: Listing syscall Entry Probes	668
Example 2: Summarize read() Returned Size	668
Example 3: Count syscalls by Process Name	670
Example 4: Count syscalls by syscall Name, for Process ID 123	671
Example 5: Count syscalls by syscall Name, for "httpd" Processes	672
Example 6: Trace File open(s) with Process Name and Path Name	672
Example 7: Summarize read() Latency for "mysqld" Processes	672
Example 8: Trace New Processes with Process Name and Arguments	673
Example 9: Sample Kernel Stacks at 100 Hz	674
References	674
Appendix F Solutions to Selected Exercises	675
Chapter 2—Methodology	675
Chapter 3—Operating Systems	675
Chapter 6—CPUs	675
Chapter 7—Memory	676
Chapter 8—File Systems	676
Chapter 9—Disks	677
Chapter 11—Cloud Computing	677

Appendix G Systems Performance Who's Who	679
Glossary	683
Bibliography	689
Index	697



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:

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



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.

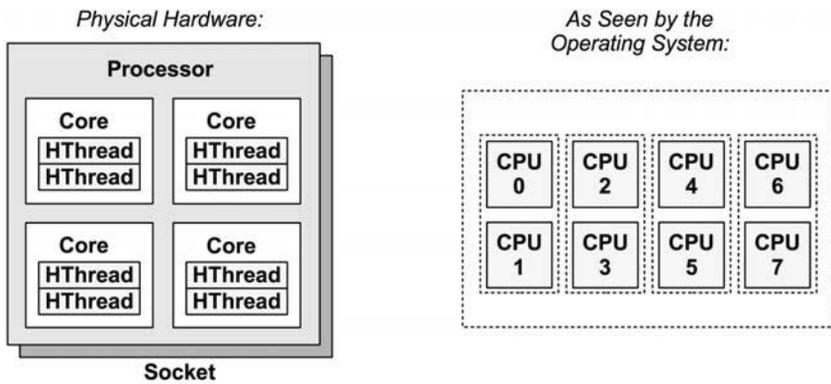


Figure 6-1 CPU architecture

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.

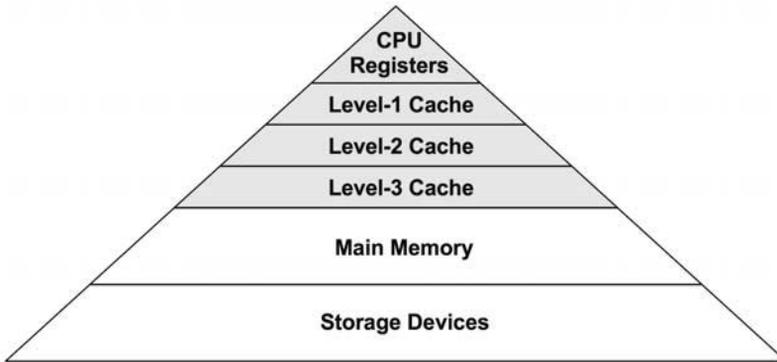


Figure 6-2 CPU cache sizes

6.2.3 CPU Run Queues

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

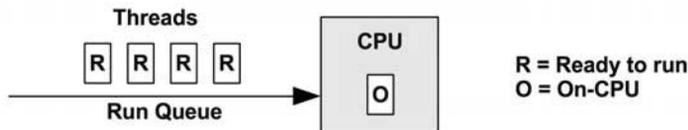


Figure 6-3 CPU run queue

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.

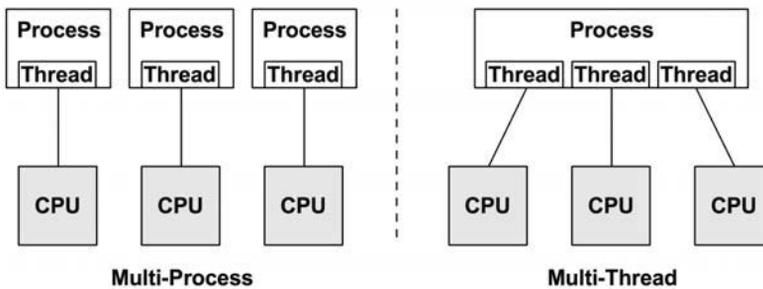


Figure 6-4 Software CPU scalability techniques

Table 6-1 Multiprocess and Multithreading Attributes

Attribute	Multiprocess	Multithreading
Development	Can be easier. Use of <code>fork()</code> .	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 <code>fork()/exit()</code> , 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 <code>exit()</code> 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.

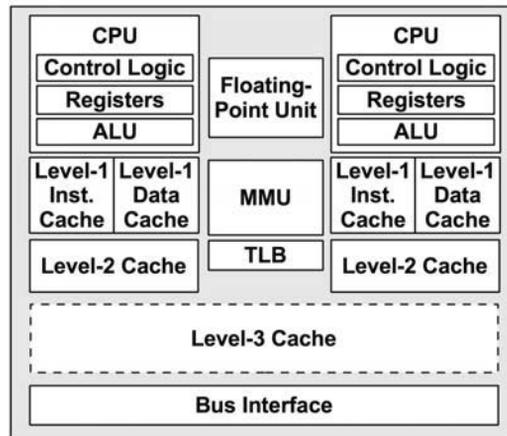


Figure 6-5 Generic two-core processor components

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 over-clocking 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

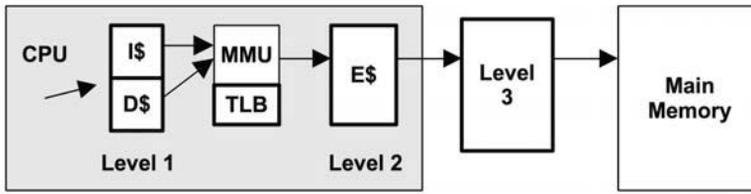


Figure 6-6 CPU cache hierarchy

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.

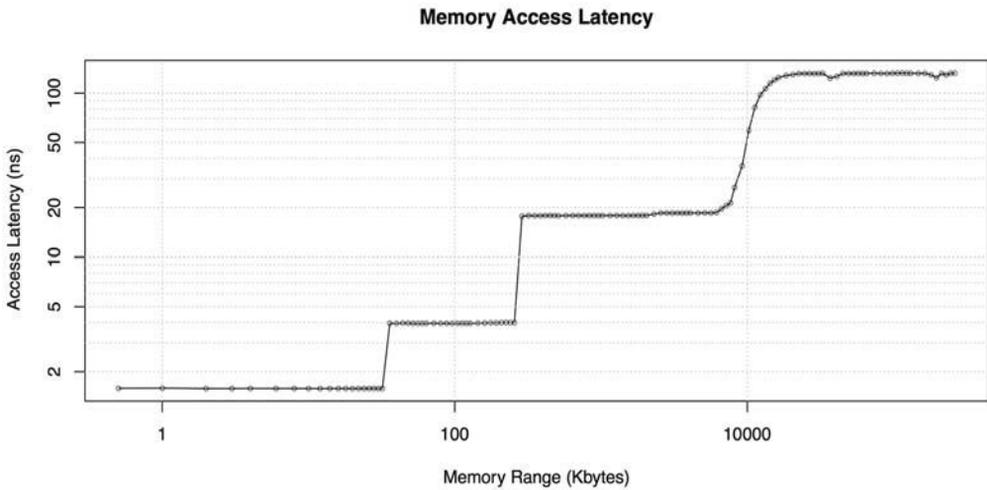


Figure 6-7 Memory access latency testing

- **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).

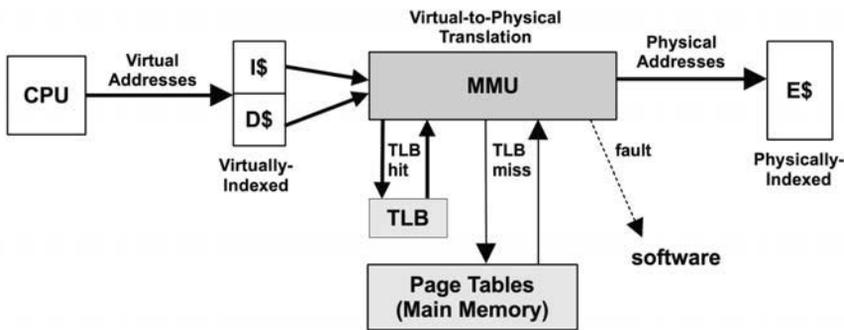


Figure 6-8 Memory management unit and CPU caches

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.

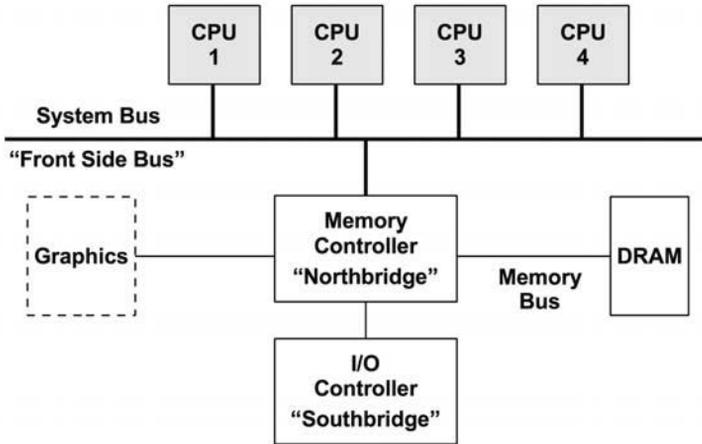


Figure 6-9 Example Intel front-side bus architecture, four-processor

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.

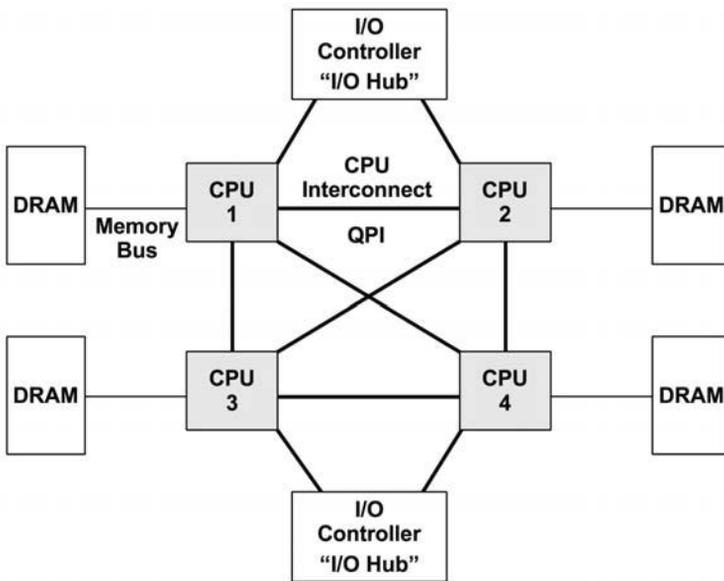


Figure 6-10 Example Intel QPI architecture, four-processor

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.

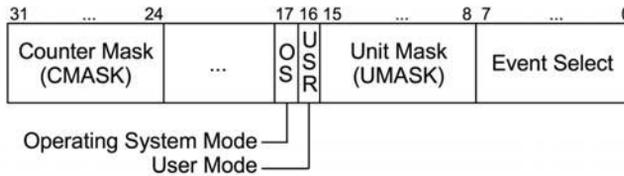


Figure 6-11 Example Intel performance event-select MSR

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.

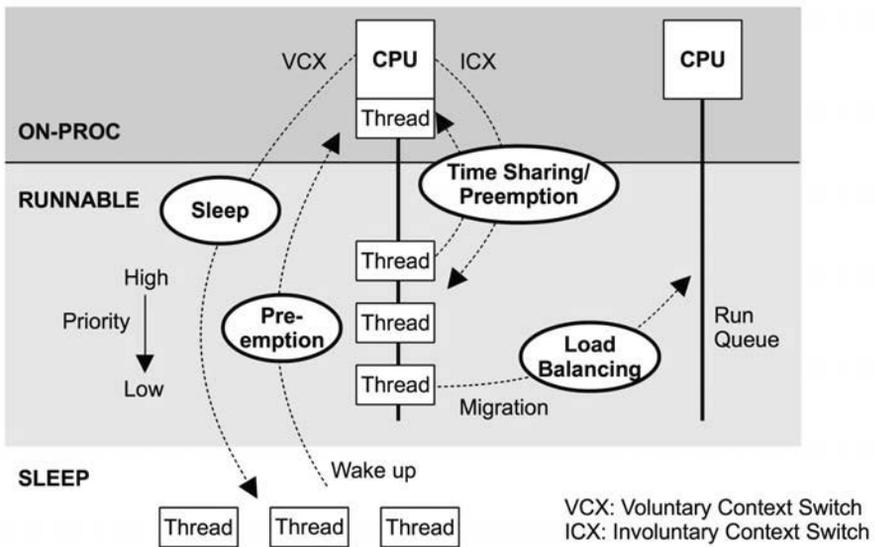


Figure 6-12 Kernel CPU scheduler functions

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.

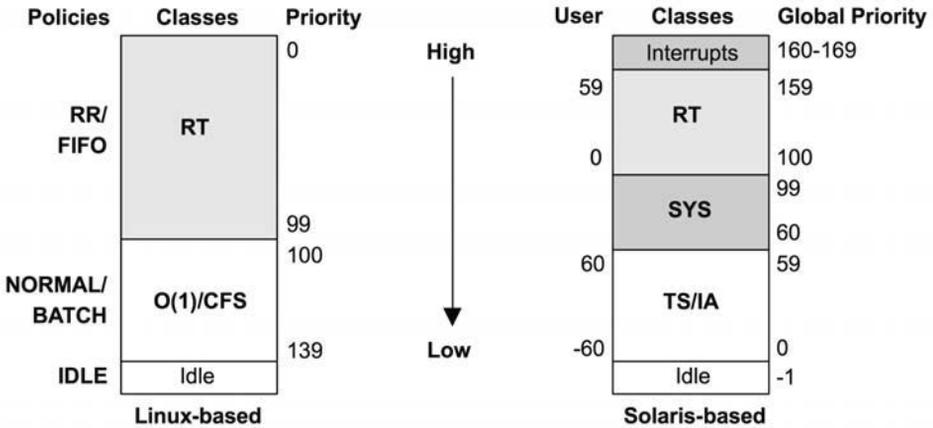


Figure 6-13 Thread scheduler priorities

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 quanta (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 $159 + \text{IPL}$ (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 *cpuset*s. 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.

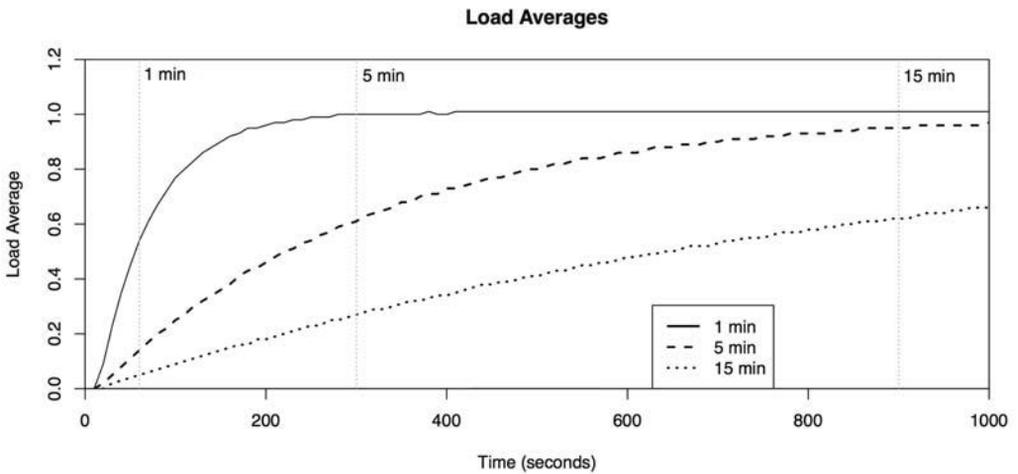


Figure 6-14 Exponentially damped load averages

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 runnable 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    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 12.50
02:47:50 0 22.00 0.00 57.00 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 16.00
```

continues

```

02:47:50  2  24.00  0.00 52.00  0.00  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  0.00
02:47:50  4 100.00  0.00  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  0.00
02:47:50  6 100.00  0.00  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  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  0.00
02:47:50  9  11.00  0.00 53.00  0.00  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  0.00
02:47:50 11  28.00  0.00 61.00  0.00  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  0.00 17.00
02:47:50 13  12.00  0.00 56.00  0.00  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  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  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 (`a11`). The columns are

- **CPU**: logical CPU ID, or `a11` 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 cpul3  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, pttime

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`0xfffffffffb800ed9
    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`__prioset+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 un-mangle 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	e	62
	1024	e	79
	2048		13
	4096		21
	8192		8
	16384		2
	32768		41
	65536	ee	3740
	131072	e	134
	262144		0

Most of the time this function took between 65 and 131 μ s 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
<code>on-cpu</code>	The current thread begins execution on-CPU.
<code>off-cpu</code>	The current thread is about to end execution on-CPU.
<code>remain-cpu</code>	The scheduler has decided to continue running the current thread.
<code>enqueue</code>	A thread is being enqueued to a run queue (examine it via <code>args[]</code>).
<code>dequeue</code>	A thread is being dequeued from a run queue (examine it via <code>args[]</code>).
<code>preempt</code>	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 μ s.

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

#	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 `cpustrack(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.

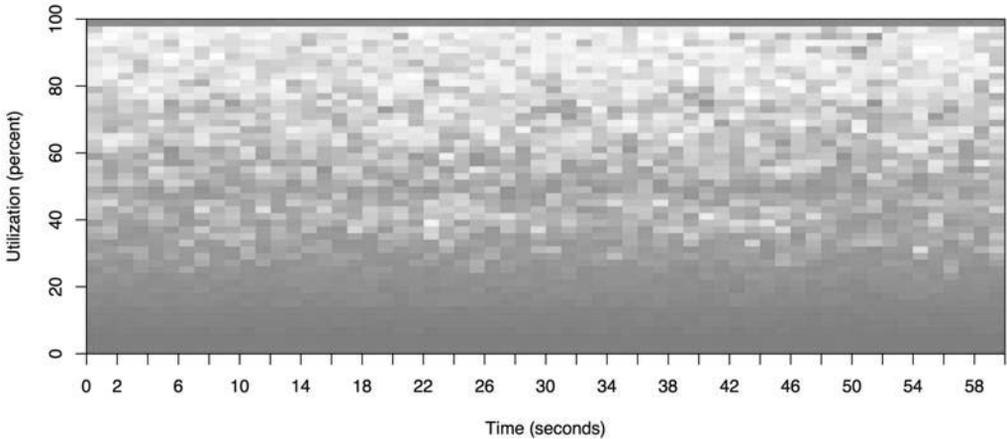


Figure 6-15 CPU utilization heat map, 5,312 CPUs

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

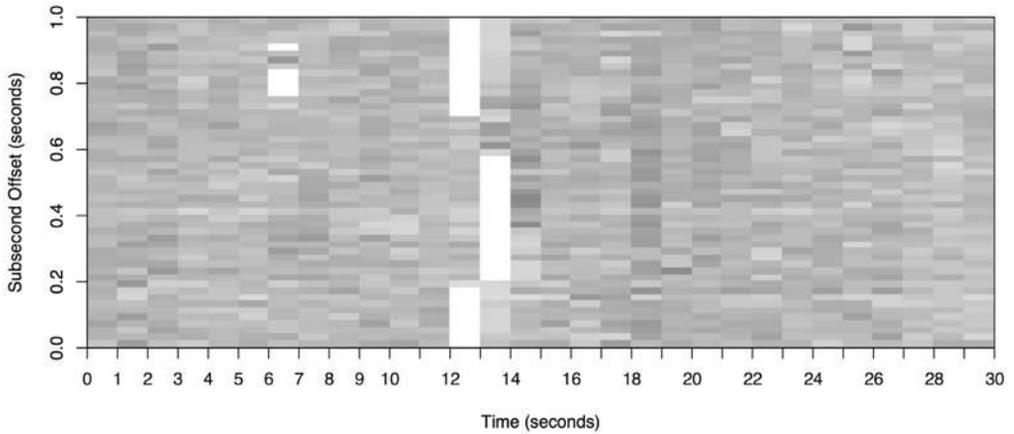


Figure 6-16 CPU subsecond-offset heat map

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).

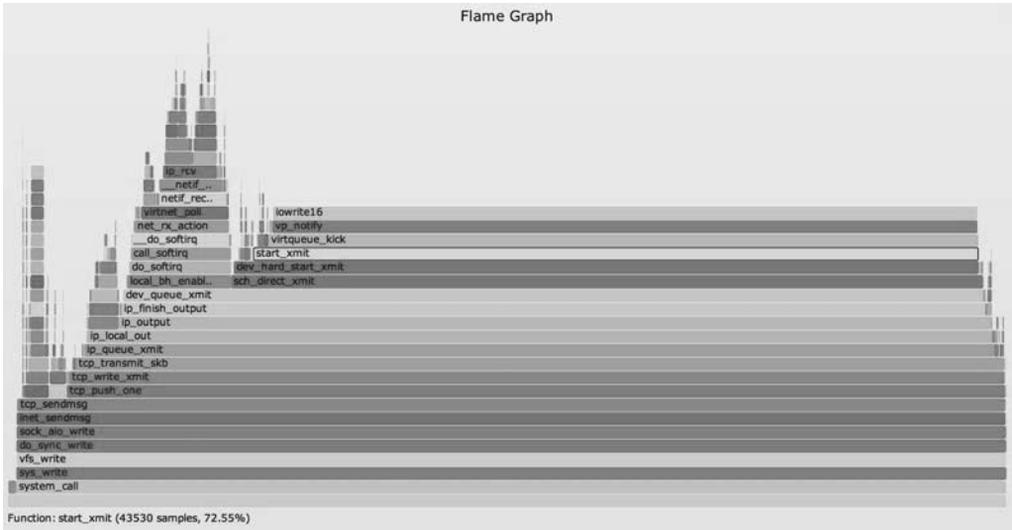


Figure 6-17 Linux kernel flame graph

- The width of the box shows the total time it was on-CPU or part of an ancestor 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 super-user 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 `prctl(1)` command. For example:

```
# prctl -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  mysql/15620
11076 mysql      3.5  1.0  0.0  0.0  0.0  42   46  7.6  1K   42  18K  0  mysql/15189
11076 mysql      3.0  0.9  0.0  0.0  0.0  34   53  8.9  1K   20  17K  0  mysql/14454
11076 mysql      3.1  0.6  0.0  0.0  0.0  55   36  5.7  729  27  11K  0  mysql/15849
11076 mysql      2.5  1.1  0.0  0.0  0.0  28   59  8.6  1K   35  19K  0  mysql/16094
11076 mysql      2.4  1.1  0.0  0.0  0.0  34   54  8.3  1K   45  20K  0  mysql/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

Numbers

- 10 GbE networking, 493
- 32-bit/64-bit architecture, word size and, 198–199

A

- Accelerated receive flow steering, Linux network stack, 493
- `accept()`
 - DTrace for socket connections, 521
 - function of key system calls, 95
- Access timestamps, 336
- Accounting
 - Linux delay accounting, 130–131
 - process accounting, 132
 - Solaris microstate accounting, 131
- ACKs (acknowledgements)
 - defined, 683
 - delayed, 486
 - duplicate ACK detection, 485
 - in TCP three-way handshake, 484
- Actions
 - DTrace, 138–139
 - SystemTap, 146
- Active benchmarking, 604–606
- Active TCP connection rate
 - in TCP three-way handshakes, 484
 - workload characterization of, 496

- Activities, performance activities in order of execution, 3–4
- Ad hoc checklists, 38–39
- Ad hoc experimentation
 - testing CPU performance, 255
 - testing disk performance, 465
 - testing file system performance, 384
- Adaptive mutex locks
 - defined, 683
 - lock analysis and, 183
 - types of synchronization primitives, 161
- Adaptive spinning mutexes, 161
- Address space, 266, 683
- Addresses, 683
- Advanced tracing. *see* Dynamic tracing
- Agents, for monitoring, 76
- Aggregation
 - converting DTrace to SystemTap, 669, 671
 - defined, 683
 - DTrace, 177
 - as variable type/overhead, 139
- Aggregation variables, DTrace, 139–141
- Algorithms
 - associativity and, 202–203
 - Big O notation analyzing complexity of, 156–158
 - cache management, 31–32
 - CSMA/CD, 487
 - elevator seeking and, 397, 410, 419, 457

Algorithms (*continued*)

- hashing, 162
- RCU-walk (read-copy-update-walk), 343
- scheduling, 212
- TCP congestion control, 102, 485–486, 538
- thread management, 99

Allocators

- improving performance of multithreaded applications, 318
- for memory usage, 272, 286–287
- for page management in Linux, 281
- tracing allocation, 308–311
- types of, 287–289

Amdahl's Law of Scalability, 60–61

Analysis

- appropriate level of, 22–23
- benchmarking and, 589–592
- drill-down analysis, 50–51

Analysis, application-related

- Big O notation applied to, 156–158
- drill-down analysis, 182
- lock analysis, 182–185
- overview of, 167–168
- syscall analysis, 173
- thread-state analysis, 168–171

Analysis, CPU-related

- checking scheduler latency, 245–246
- DTrace and, 236–238
- examining CPC (`cpustat`), 249–250
- Linux performance tools in, 250
- load averages and, 224–226
- monitoring top running processes, 231–234
- multiprocessor statistics (`mpstat`), 227–229
- overview of, 224
- `perf` analysis of, 243–244
- `perf` documentation, 249
- printing CPU usage by thread or process (`pidstat`), 234
- profiling kernel with DTrace, 236–237
- profiling processes with `perf`, 245
- profiling system with `perf`, 244–245
- profiling user with DTrace, 238–240
- reporting on CPU usage (`time`, `ptime`), 235–236
- reporting on system activity (`sar`), 230
- Solaris performance tools, 251
- summarizing CPU cycle behavior with `perf stat`, 246–247
- SystemTap analysis of, 243
- tracing CPU cross calls, 241
- tracing functions, 240–241
- tracing interrupts, 241–242
- tracing scheduler tracing, 242–243

- tracing software with `perf`, 247–249
- uptime analysis, 224–225
- viewing process status (`ps`), 230–231
- virtual memory statistics (`vmstat`), 226–227
- visualizations, 251–254

Analysis, disk-related

- counting I/O requests, 448–449
- of disk controllers, 459–460
- dynamic tracing, 450
- event tracing, 444
- event tracing applied to SCSI events, 449–450
- examining events with DTrace, 442–443
- heat maps, 462–465
- io provider (DTrace) for visibility of block device interface, 443–444
- line charts/graphs, 461
- overview of, 431–432
- `perf`, 451–452
- printing statistics with `pidstat`, 441–442
- reporting/archiving current activity (`sar`), 440–441
- scatter plots, 462
- summarizing I/O seek distances, 445–446
- summarizing I/O size, 444–445
- summarizing latency, 446–447
- summarizing per-disk statistics (`iostat`), 432
- summarizing per-disk statistics on Linux, 433–436
- summarizing per-disk statistics on Solaris, 436–440
- SystemTap, 451
- top analysis of (`disktop.stp`), 454
- top analysis of (`iostat`), 452–454
- tracing block device events on Linux (`blktrace`), 457–459
- tracing with `iosnoop`, 455–457
- viewing health statistics with `smartctl`, 460–461
- visualizations, 461

Analysis, file system-related

- analyzing file system cache with, 376–377
- analyzing file system statistics with `sar`, 377–378
- analyzing file systems (`kstat`), 381–382
- analyzing kernel memory usage on Solaris systems, 379
- analyzing kernel slab caches related to file systems, 378–379
- of block device I/O stacks, 370–371
- debugging syscall interface on Linux, 364–365
- DTrace and, 365
- dynamic tracing, 373–375
- file system statistics, 364

- LatencyTOP tool, 375
- measuring system latency with DTrace, 367–368
- measuring VFS latency with DTrace, 368–370
- memory summary using `mdb::memstat`, 380–381
- memory summary with `/proc/meminfo`, 380
- operation counts with DTrace, 365–366
- other tools, 382–383
- overview of, 362–363
- printing cache activity on UFS using `fcachestat`, 379–380
- showing statistics with `free`, 376
- summarizing event counts with DTrace, 367
- SystemTap analysis of, 375
- tracing slow events, 372–373
- tracing ZFS read latency, 371–372
- of virtual file system statistics, 363–364
- visualizations, 383
- Analysis, memory-related
 - allocation tracing, 308–311
 - fault tracing, 311–312
 - kernel memory usage on Solaris systems (`kmastat`), 302–303
 - list of performance tools, 312–314
 - mapping process memory (`pmap`), 306–308
 - overview of, 295
 - page-out daemon tracing, 312
 - process status (`ps`), 304–305
 - slab cache usage (`slabtop`), 302
 - system activity reporters (`sar`), 298–301
 - SystemTap, 312
 - top running processes on Linux (`top`), 305
 - top running processes on Solaris (`prstat`), 305–306
 - virtual memory statistics (`vmstat`), 295–298
- Analysis, network-related
 - capturing and inspecting packets (`snoop`), 517–520
 - capturing and inspecting packets (`tcpdump`), 516–517
 - configuring network interfaces and routes (`ip`), 512
 - configuring network interfaces (`ifconfig`), 511
 - DTrace for examining network events, 520–533
 - experimental tools, 535–536
 - other Linux and Solaris tools, 534
 - `perf` for static and dynamic tracing of network events, 533–534
 - printing network interface statistics (`nicstat`), 512–513
 - providing interface statistics on Solaris (`dladm`), 513–514
 - reporting on network activity (`sar`), 509–511
 - reporting on network statistics (`netstat`), 503–509
 - SystemTap for tracing file system events, 533
 - testing network connectivity (`ping`), 514
 - testing routes and bandwidth between routes (`pathchar`), 515–516
 - testing routes (`traceroute`), 514–515
 - Wireshark tool for packet capture and analysis, 520
- Anonymous memory, 266
- Anonymous page-ins, 624, 626–627
- Anonymous paging, 268–269
- Anticipatory I/O latency policy, 420
- Anti-methodologies
 - blame-someone-else, 38
 - random change, 37
 - streetlight, 36–37
- API, 683
- Application calls, tuning file systems and, 387–388
- Application servers, in cloud architecture, 547
- Applications
 - basics of, 153–155
 - breakpoint tracing on Linux (`strace`), 173–175
 - breakpoint tracing on Solaris (`truss`), 176–177
 - buffered tracing, 177–179
 - buffering, 159
 - caching, 158
 - compiled languages and, 164–165
 - concurrency and parallelism, 160–162
 - CPU profiling, 171–173
 - disk I/O vs. application I/O, 407
 - exercises and references, 186–187
 - garbage collection, 166–167
 - interpreted languages, 165–166
 - I/O profiling, 180–181
 - I/O sizing, 158–159
 - methodologies, 167–168
 - non-blocking I/O, 162–163
 - observability eliminating unnecessary work, 156
 - optimizing common code path, 156
 - overview of, 153
 - performance objectives, 155–156
 - polling, 159–160
 - processor binding, 163
 - programming languages and, 163–164
 - static performance tuning, 185–186
 - tuning targets, 22

Applications (*continued*)

- USE method applied to, 181–182
 - virtual machines and, 166
 - workload characterization applied to, 181
- apprtrace, for investigating library calls on Solaris, 182

Architecture

- cloud computing, 547–548
- load vs. architecture in analysis of performance issues, 24

Architecture, CPU-related

- cache, 200–204
- CPCs (CPU performance counters), 206–208
- hardware, 199
- idle threads, 213
- interconnects, 204–206
- MMU (memory management unit), 204
- NUMA groups, 214
- overview of, 191, 199
- processor resource-aware, 214
- processors, 199–200
- schedulers, 209–210
- scheduling classes, 210–213
- software, 209

Architecture, disk-related

- interfaces, 414
- magnetic rotational disks, 408–411
- operating system disk I/O stack, 418–421
- overview of, 407
- solid-state disks, 411–413
- storage types, 415–418

Architecture, file system-related

- btrfs, 351
- buffer cache, 339–341
- cache types, 339
- COW (copy-on-write) strategy, 344
- dentry (Dcache) cache, 343
- DNLN (directory lookup cache), 341
- ext3 and ext 4, 347–348
- features, 344–345
- FFS, 345–347
- inode cache, 343
- I/O stack model, 337
- overview of, 337
- page cache, 340–343
- types of, 345
- UFS (Unix File System), 347
- VFS (virtual file system), 337–338
- volumes and pools, 351–352
- ZFS, 348–351

Architecture, memory-related

- allocators, 286–289
- busses, 274–276

- free lists, 280–281
- hardware, 273
- heap growth and, 286
- main memory, 273–274
- methods for freeing memory, 278–280
- MMU (memory management unit), 276–277
- overview of, 272–273
- page scanning for freeing memory, 282–284
- process address space, 284–285
- reaping for freeing memory, 281–282
- software, 278

Architecture, network-related

- hardware, 486–488
- protocols, 483–486
- software, 488–493

Arguments, DTrace, 137

argumentum ad populum logic, 597

Arithmetic mean, 70

Array, 683

The Art of Capacity Planning (Allspaw), 66

Associative arrays

- converting DTrace to SystemTap, 665–666, 670
- defined, 683
- I/O latency and, 447
- SCSI events and, 449
- socket I/O and, 523
- as variable type/overhead, 138–140, 143

Associativity, CPU cache and, 202–203

Asynchronous operations, disk I/O and, 407

AT&T (American Telephone and Telegraph Company), 683

Available swap, virtual memory and, 271–272

Averages

- over time, 71
- plotting statistical values, 77–78
- types of, 70–71

B

Backlog

- connection, 481
- tracing backlog drops with DTrace, 529–531

Backlog queues, TCP

- analysis of usage, 500
- overview of, 492–493
- tuning on Linux, 537–538
- tuning on Solaris, 541

Balanced system, 683

Balloon driver, resource controls in hardware virtualization, 573–574

Bandwidth

- defined, 396
- interface types based on, 487

- networking terminology, 474
- resource controls for network limits, 502
- Baseline statistics, 54
- Benchmarking
 - active benchmarking, 604–606
 - activities, 588–589
 - analysis, 589–590
 - applying micro-benchmarking to file systems, 361–362
 - casual approach to, 591
 - changing multiple factors in, 594–595
 - cheating, 597
 - cloud computing use in simulation, 57–58
 - complex tools for, 592
 - CPU profiling, 606–607
 - custom benchmarks, 608
 - exercises and references, 614–616
 - faith issues, 591
 - friendly fire, 595
 - ignoring errors and variance, 593
 - ignoring perturbations, 593–594
 - industry-standard benchmarks, 601–602
 - methodologies, 602–603
 - micro-benchmarking, 56–57, 502–503, 597–599
 - misleading benchmarks, 595–596
 - numbers without analysis, 591–592
 - overview of, 587
 - passive benchmarking, 603–604
 - ramping load, 608–611
 - replay, 600
 - sanity check, 611
 - simulation, 599–600
 - sins, 596–597
 - statistical analysis, 612–613
 - SysBench tool, 255
 - testing effect of software change, 12
 - testing wrong thing, 592–593
 - USE method, 607–608
 - workload characterization, 608
 - workload characterization as input for, 49–50
- BGSAVE configuration, 631–633
- Big O notation, applied to performance analysis, 156–158
- Bimodal distributions, in statistics, 73
- Binaries, in compiled languages, 164
- Binary translation, 566–567
- Binding
 - CPU binding, 222
 - process binding, 259
 - processor binding, 163
- BIOS, tuning, 260
- Blame-someone-else anti-method, 38
- blktrace
 - system-wide tracing, 118
 - tracing block device events on Linux, 457–459
- Block devices
 - analyzing block device I/O stacks, 370–371
 - device drivers for, 103
 - interface, 418–421
 - io provider (DTrace) for visibility of block device interface, 443–444
 - tracing events on Linux (blktrace), 118, 457–459
- Block interface, 103
- Block store, for persistent storage, 550
- Bonnie/Bonnie++ benchmarking tool, 385, 604–606
- Borrowed virtual time (BVT), hypervisor CPU scheduler, 572
- Bottlenecks
 - checking resource bottlenecks, 10
 - complexity of systems and, 5
 - defined, 16
 - lock analysis and, 182
 - resource limits and, 66–67
 - USE method for identifying, 422
- Breakpoint tracing
 - on Linux (strace), 173–175
 - on Solaris (truss), 176–177
- brk(), system calls, 96
- Broadcast messages, 476–477
- BSD (Berkeley Software Distribution)
 - defined, 683
 - memory management, 98
 - resource controls, 104
- btrfs file system, 351
- Buckets
 - hash table, 162
 - heat map ranges, 79
- Buffers/buffering
 - block devices and buffer cache, 103
 - buffer bloat, 481
 - buffer cache, 103, 339–341
 - buffer size and performance trade-offs, 21
 - buffered tracing, 177–179
 - connection backlog, 481
 - defined, 684
 - for improving application performance, 159
 - networks and, 481
 - ring, 493
 - TCP, 492–493
 - tuning Linux socket and TCP, 537
 - tuning on Solaris, 540

Bursting

- dynamic sizing in cloud with, 549
- resource controls in OS virtualization, 556

Busses, memory, 274–276

BVT (borrowed virtual time), hypervisor CPU scheduler, 572

Bytecode, virtual machine instruction set, 166

Bytes, I/O size in, 403

C

C programming language, 684

Cache/caching

- algorithms for cache management, 31–32
- analyzing file system cache with `top`, 376
- analyzing file system cache with `vmstat`, 376–377
- analyzing kernel slab caches, 378–379
- buffer cache, 103
- cache coherency, 158, 203–204
- cache hit, 684
- cache layers for disk I/O, 101–102
- cache line size, 203
- cache miss, 31, 684
- cache tuning, 360
- cache warmth, 192, 684
- CPU memory cache, 191–192
- defined, 16
- disk cache, 397–398
- disk I/O performance and, 401–402
- file system cache, 11, 271, 325–326
- file systems, 327–328, 339–343
- flushing system cache, 387
- hot, cold, and warm caches, 32
- for improving application performance, 158
- overview of, 30–31
- printing cache activity on UFS using `fcachestat`, 379–380
- RAID and, 417
- second-level cache, 326
- tuning, 55–56
- tuning disk cache, 429
- write-back caching, 330

`cachegrind` profiling tool, 119

Callouts, scheduled kernel routines, 88

Capacity, file system, 337

Capacity planning

- activities in systems performance, 2
- benchmarking during, 588
- cloud computing and, 8, 548–549
- factor analysis, 68
- overview of, 65–66

- resource analysis and, 33
- resource limits and, 66–67
- scaling solutions and, 69, 223, 431

Capacity resources, main memory as, 45

Capacity-based utilization, 28–29

Caps, CPU, 556

Carrier sense multiple access with collision detection (CSMA/CD), 487

CAS (column address strobe), in measuring access time of main memory, 273

Cascading failures, system complexity and, 5

Case study. *see* Redis application, as troubleshooting case study

Casual benchmarking issues, 591

CFQ (completely fair scheduler), I/O latency policies, 420

cgroups (control groups), resource management options in Linux, 104, 318

Challenges, in performance engineering

- complexity of systems, 5–6
- multiplicity of performance issues, 6
- overview of, 4
- subjective nature of performance, 5

Change management, software change case study, 11–13

Character interface, device drivers providing, 103

Cheating, benchmarking and, 597

`chroot`, 552

Circular buffers, 159

Clause-local variables, DTrace, 139

Client, 684

`clock()` routine, kernel and, 88–89

Clock rate, CPUs, 193

`close()`, system calls, 95

Cloud Analytics. *see* Joyent Cloud Analytics

Cloud Analytics, Joyent, 81

Cloud API, 546

Cloud computing

- background, 546
- benchmarking in, 588–589
- capacity planning, 548–549
- comparing technologies, 581–583
- defined, 546
- exercise solutions, 677
- exercises and references, 583–585
- hardware virtualization. *see* Hardware virtualization
- multitenancy, 550–551
- OS virtualization. *see* OS virtualization
- overview of, 8–9, 545–546
- price/performance ratio, 546
- scalable architecture, 547–548

- simulating enterprise systems using, 57–58
 - storage, 550
 - USE method and, 48
- Cloudy day performance testing, benchmarks for, 599
- Code, optimizing common code path in applications, 156
- Coefficient of variation (CoV), 72
- Coherence, visualization of scalability profiles, 59
- Cold cache, 32
- Collisions, interface identifying, 487
- Column address strobe (CAS), in measuring access time of main memory, 273
- Command, 684
- Comparisons of virtualization technologies, 581–583
- Compatible Time-Sharing System (CTSS), 684
- Competition, benchmarking the, 594–595
- Compiled languages, 164–165
- Compilers
 - CPU compiler options, 256
 - improving CPU runtime, 199
 - optimizing, 164–165
- Completely fair scheduler (CFQ), I/O latency policies, 420
- Completion, targets for workload analysis, 34
- Complexity of systems, 5–6
- Concurrency
 - application performance and, 160–162
 - defined, 684
- Configuration, tuning network performance, 542
- Congestion avoidance, 483, 485
- Congestion control, TCP
 - algorithms for, 485–486
 - defined, 483
 - tuning on Linux, 538
- Congestion window
 - analysis of size, 500
 - defined, 483
- connect (), system calls, 95, 520–522
- Connect latency, networks, 480, 524
- Connection backlog, networks, 481
- Connection life span, networks, 481
- Connections
 - local network, 482–483
 - performance monitoring of TCP, 498
- Contention, visualization of scalability profiles, 59
- Context switch
 - defined, 86
 - between user mode and kernel mode, 89
- Control unit (control logic), at heart of CPU, 200
- Controllers, disk. *see* Disk controllers
 - Controllers, network
 - as networking hardware component, 487
 - overview of, 475
 - USE method for analyzing, 496
- Copy-on-write (COW) strategy, 93, 344
- Core, 190
- Counters
 - CPCs (CPU performance counters), 206–208
 - CPU performance, 131
 - overview of, 116
 - system-wide and per-process, 117
- CoV (coefficient of variation), 72
- COW (copy-on-write) strategy, 93, 344
- CPCs (CPU performance counters)
 - checking memory bus load, 293
 - CPU cycle analysis, 219–220
 - tools for examining, 249–250
 - types of CPU hardware, 206–208
- CPI (cycles per instruction)
 - CPU cycle analysis, 219–220
 - CPU metrics, 194–195
 - PICs (performance instrumentation counters) and, 249–250
- CPU affinity, 163, 192
- CPU cache
 - associativity and, 202–203
 - cache coherency, 203–204
 - cache line size, 203
 - for improving memory access performance, 276
 - latency and, 202
 - overview of, 200–202
- CPU counters. *see* CPCs (CPU performance counters)
- CPU cross calls
 - defined, 684
 - DTrace analysis of, 241
 - preemption and, 103
- CPU cycles
 - clock rate and, 193
 - defined, 684
- CPU instructions
 - CPI (cycles per instruction)/IPC (instructions per cycle) metrics, 194–195
 - defined, 190
 - instruction pipeline, 194
 - instruction sets, 193–194
 - instruction width, 194
 - micro-benchmarking, 222
- CPU mode
 - analysis of time in user mode or kernel mode, 196
 - determining CPU mode of application, 154

- CPU mode (*continued*)
 - kernel mode, 88–89
 - user mode, 89
- CPU profiling, 171–173, 606–607
- CPUs (central processing units)
 - affinity, 163, 192
 - analysis, 214, 224
 - architecture, 191, 199
 - associativity and, 202–203
 - binding, 222
 - cache. *see* CPU cache
 - checking resource controls, 562
 - clock rate, 193
 - comparing virtualization technology
 - performances, 581
 - compiler optimization, 199
 - compiler options, 256
 - counters. *see* CPCs (CPU performance counters)
 - cpuset, 259
 - cpustat, 249–250
 - cross calls, 103, 684
 - cycle analysis, 219–220
 - cycles, 193, 684
 - defined, 684
 - determining CPU mode, 154
 - exercise solutions, 675–676
 - exercises and references, 260–263
 - experimentation, 254–255
 - factor analysis, 68
 - garbage collection and, 167
 - hardware, 199
 - idle threads, 213
 - instructions. *see* CPU instructions
 - interconnects, 204–206
 - I/O wait as performance metric, 406
 - latency and, 202
 - Linux performance tools, 250
 - load averages in analyzing, 224–226
 - memory cache, 191–192
 - micro-benchmarking, 222–223
 - MMU (memory management unit), 204
 - mode. *see* CPU mode
 - multiprocessing/multithreading, 197–198
 - multiprocessor statistics (mpstat), 227–229
 - multiprocessor support, 103
 - NUMA groups, 214
 - overhead in hardware virtualization, 566–569
 - overhead in OS virtualization, 553
 - overview of, 189–190
 - parallelism and, 160
 - performance counters, 131
 - performance monitoring, 220
 - performance trade-offs between memory and, 21
 - pidstat analysis, 234
 - preemption, 196
 - price/performance ratio in cloud computing, 546
 - priority inversion, 196–197
 - priority tuning, 221–222
 - process binding, 259
 - processor binding, 163
 - processor options, 260
 - processors and, 199–200
 - profiling, 171–173, 218–219, 606–607
 - prstat analysis, 232–234
 - ps analysis, 230–231
 - resource controls, 222, 260, 556–557
 - run queues, 192–193
 - sar analysis, 230
 - saturation and, 196
 - scaling, 223–224
 - scheduler class tuning, 258
 - schedulers, 98–99, 209–210, 257–258
 - scheduling classes, 210–213
 - scheduling priority and class, 256–257
 - software, 209
 - Solaris performance tools, 251
 - static performance tuning, 220–221
 - surface plots for per-CPU utilization, 80–81
 - SystemTap analysis of, 243
 - terminology regarding, 190
 - time, ptime analysis, 235–236
 - time scales of CPU cycles, 20
 - tools method, 215
 - top analysis, 231–232
 - tuning, 214, 256
 - uptime analysis, 224–225
 - USE method, 216, 623
 - utilization measurement, 195
 - virtual CPUs. *see* vCPUs (virtual CPUs)
 - visualizations, 251–254
 - vmstat analysis, 226–227
 - word size, 198–199
 - workload characterization applied to, 216–218
- CPUs, DTrace analysis of
 - cross calls, 241
 - function tracing, 240–241
 - interrupts, 241–242
 - kernel profiling, 236–237
 - one-liners, 237–238
 - overview of, 236
 - scheduler tracing, 242–243
 - user profiling, 238–240

CPUs, perf analysis of
 checking scheduler latency, 245–246
 documentation, 249
 overview of, 243–244
 process profiling, 245
 software tracing, 247–249
 stat command for summarizing CPU cycle
 behavior, 246–247
 system profiling, 244–245

cpusets, creating CPU sets in Linux, 259

cpusets, Linux, 222

cpustat, for system-wide analysis, 249–250

cputrack, for process analysis, 249

CR3 profiling, for observability in hardware
 virtualization, 579–580

Credit-based hypervisor CPU scheduler, 572

Cross calls, CPU
 defined, 684
 DTrace analysis of, 241
 preemption and, 103

CSMA/CD (carrier sense multiple access with
 collision detection), 487

CTSS (Compatible Time-Sharing System), 684

Custom benchmarks, 608

CV (coefficient of variation). *see* CoV (coefficient
 of variation)

Cycles, CPU
 CPI (cycles per instruction) metric, 194–195
 cycle analysis, 219–220, 293
 resource controls, 222
 summarizing CPU cycle behavior, 246–247

Cycles per instruction. *see* CPI (cycles per
 instruction)

Cyclic page cache, Solaris methods for freeing
 memory, 279

D

D programming language, 137, 684

Data rate, throughput and, 16

Database servers, in cloud architecture, 547

Databases
 performance tuning targets, 22
 sharding, 69

Datagrams, sending with UDP, 486

Dcache (Dentry) cache, 343

DDR SDRAM (double data rate synchronous
 dynamic random-access memory), 274

Deadlines, I/O latency policies, 420

debuginfo file, 245, 684

DEC (Digital Equipment Corporation), 684

Decayed averages, 71

Degradation of performance, for nonlinear
 scalability, 25–26

Delay accounting, Linux, 130–131, 170

Delayed ACKs algorithm, for TCP congestion
 control, 486

Demand paging, 269–270

Denial-of-service (DoS) attacks, 49

Dentry (Dcache) cache, 343

Development, benchmarking during, 588

Device backlog queue, tuning on Linux, 538

Device drivers, 103

Device I/O, 574

df, analyzing file systems, 382

Diagnosis cycle, 41

Digital Equipment Corporation (DEC), 684

Direct I/O file systems, 331–332

Directory lookup cache (DNLC), 341

Directory organization, 99–100

Disk controllers
 analyzing with MegaCli, 459–460
 defined, 684–685
 magnetic rotational disks, 411
 micro-benchmarking, 430
 overview of, 398–399
 scaling solutions, 431
 SSDs (solid-state disks), 412–413
 tuning with MegaCli, 469–470
 USE method for checking, 423

Disk devices
 storage devices, 415
 tuning, 469
 USE method for checking, 422–423

Disk heads, on hard disks, 408

Disk I/O. *see also* I/O (input/output)
 analyzing block device I/O stacks, 370–371
 vs. application I/O, 407
 cache, 401–402
 cache layers, 101–102
 characterizing disk I/O workload, 424–425
 counting I/O requests (I/O stack frequency),
 448–449
 event tracing, 427–428, 442–444
 I/O wait, 406
 IOPS (I/O per second), 404
 latency, 399–401
 micro-benchmarking, 361–362, 429–430
 observability in OS virtualization, 561
 operating system disk I/O stack, 418
 printing statistics with pidstat, 441–442
 random vs. sequential, 402–403
 resource controls, 558, 563
 scatter plots, 78

Disk I/O (*continued*)

- simple model for I/O requests, 397
 - sizing, 403–404
 - slow disk case study, 9–11
 - summarizing disk I/O size, 444–445
 - summarizing I/O seek distances, 445–446
 - summarizing I/O size, 444–445
 - summarizing latency, 446–447
 - summarizing per-disk statistics (*iostat*), 432
 - summarizing per-disk statistics on Linux, 433–436
 - summarizing per-disk statistics on Solaris, 436–440
 - time measurements (response time, service time, and wait time), 399–400
 - time scales, 400–401
 - top analysis of (*disktop.stp*), 454
 - top analysis of (*iotop*), 452–454
 - tracing with (*iosnoop*), 455–457
 - USE method applied to, 625–626
 - workload characterization, 424–426
- Disk offset, random vs. sequential I/O and, 402–403
- Disks
- analysis of, 353, 431–432
 - analyzing disk controllers with *MegaCli*, 459–460
 - architecture, 407
 - block device interface, 418–421
 - caching, 397–398
 - command for non-data transfer, 404
 - controller for, 398–399
 - counting I/O requests (I/O stack frequency), 448–449
 - defined, 684
 - disk I/O caches, 401–402
 - disk I/O vs. application I/O, 407
 - dynamic tracing, 450
 - event tracing, 427–428
 - event tracing applied to disk I/O, 444
 - event tracing applied to SCSI events, 449–450
 - examining disk I/O events with *DTrace*, 442–443
 - exercise solutions, 677
 - exercises and references, 470–472
 - experimentation, 465–467
 - heat maps, 462–465
 - interfaces, 414
 - io provider (*DTrace*) for visibility of block device interface, 443–444
 - I/O sizing, 403–404
 - I/O wait, 406
 - IOPS (I/O per second), 404
 - latency analysis, 426–427
 - line charts/graphs, 461
 - magnetic rotational disks, 408–411
 - methodologies, 421
 - micro-benchmarking, 429–430
 - NAS (network-attached storage), 417–418
 - operating system disk I/O stack, 418
 - overview of, 395–396
 - perf* analysis of, 451–452
 - performance monitoring, 423–424
 - printing disk I/O statistic with *pidstat*, 441–442
 - RAID architecture, 415–417
 - random vs. sequential I/O, 402–403
 - read/write ratio, 403
 - reporting/archiving current activity (*sar*), 440–441
 - resource controls, 429
 - saturation, 405–406
 - scaling solutions, 431
 - scatter plots, 462
 - simple model for I/O requests, 397
 - solid-state disks, 411–413
 - static performance tuning, 428–429
 - storage arrays, 417
 - storage devices, 415
 - storage types, 415
 - storing file system content, 100
 - summarizing disk I/O latency, 446–447
 - summarizing disk I/O seek distances, 445–446
 - summarizing disk I/O size, 444–445
 - summarizing per-disk I/O statistics (*iostat*), 432
 - summarizing per-disk I/O statistics on Linux, 433–436
 - summarizing per-disk I/O statistics on Solaris, 436–440
 - synchronous vs. asynchronous operation and, 407
 - SystemTap*, 451
 - terminology regarding, 396
 - time measurements (response time, service time, and wait time), 399–400
 - time scales for disk I/O, 400–401
 - tools method, 422
 - top analysis of disk I/O (*disktop.stp*), 454
 - top analysis of disk I/O (*iotop*), 452–454
 - tracing block device events on Linux (*blktrace*), 457–459
 - tracing disk I/O with (*iosnoop*), 455–457
 - tunables of disk controllers, 469–470
 - tunables of disk devices, 469
 - tunables of operating system, 467–469

- tuning, 467
- tuning cache, 429
- types of, 408
- USE method, 422–423
- utilization, 404–405
- viewing health statistics with `smartctl`, 460–461
- visualizations, 461
- workload characterization, 424–426
- `disktop.stp`, for top analysis of disks with SysTap, 454
- Dispatcher-queue latency, 192
- Distribution, of file system latency, 383
- Distribution of data
 - multimodal distributions, 73–74
 - standard deviation, percentiles, and medians, 72
- `dladm`
 - for network analysis, 513–514
 - replacing `ifconfig` on Solaris, 511
- DNL (directory lookup cache), 341
- DNS latency, 19
- Documentation/resources
 - DTrace, 143–144
 - SystemTap, 149
- Dom0, Xen, 557
- Domains, Xen, 557
- DoS (denial-of-service) attacks, 49
- Double data rate synchronous dynamic random-access memory (DDR SDRAM), 274
- Double-hull virtualization, 572
- DRAM (dynamic random-access memory)
 - as common type of main memory, 273
 - defined, 685
- Drill-down analysis
 - analyzing and tuning applications, 182
 - overview of, 50–51
 - reasons to perform, 500–501
- DTrace
 - actions, 138–139
 - advanced observability for KVM with, 578–579
 - analysis phase of drill-down analysis, 51
 - arguments, 137
 - built-in variables, 137–138
 - cloud-wide analysis tool, 81
 - CR3 profiling in, 579–580
 - D language and, 137
 - documentation and resources, 143–144
 - DTraceToolkit, 143
 - dynamic tracing with, 7–8
 - overhead, 143
 - overview of, 133–134
 - probes, 135–136
 - profiling tools, 119
 - providers, 136
 - scripts, 141–143
 - static and dynamic tracing with, 134
 - system-wide tracing, 118
 - variable types, 139–141
- DTrace, analyzing applications
 - buffered tracing, 177
 - CPU profiling, 171–173
 - drill-down analysis, 182
 - I/O profiling, 180–181
- DTrace, analyzing CPUs
 - cross calls, 241
 - function tracing, 240–241
 - interrupts, 241–242
 - kernel profiling, 236–237
 - one-liners, 237–238
 - overview of, 236
 - profiling, 218–219
 - scheduler tracing, 242–243
 - user profiling, 238–240
- DTrace, analyzing disks
 - counting I/O requests (I/O stack frequency), 448–449
 - dynamic tracing, 450
 - event tracing, 444
 - event tracing applied to SCSI events, 449–450
 - examining disk I/O events, 442–443
 - io provider for visibility of block device interface, 443–444
 - summarizing disk I/O seek distances, 445–446
 - summarizing disk I/O size, 444–445
 - summarizing disk latency, 446–447
- DTrace, analyzing file systems
 - advanced tracing, 373–375
 - block device I/O stacks, 370–371
 - measuring system latency, 367–368
 - measuring VFS latency, 368–370
 - operation counts, 365–367
 - overview of, 365
 - summarizing event counts, 367
 - tracing slow events, 372–373
 - tracing ZFS read latency, 371–372
- DTrace, analyzing memory
 - allocation tracing, 308–311
 - fault tracing, 311–312
 - for tracing page-out daemon, 312
- DTrace, analyzing networks
 - advanced network tracking scripts, 531–533
 - backlog drops, 529–531
 - network providers, 520–521
 - overview of, 495

DTrace, analyzing networks (*continued*)
 packet transmission, 527–528
 retransmit tracing, 528–529
 socket connections, 521–523
 socket internals, 525
 socket I/O, 523–524
 socket latency, 524–525
 TCP events, 525–526

DTrace, analyzing Redis application
 interrogating kernel, 629–631
 measuring file system syscalls, 628–629
 measuring read latency, 627
 selecting kernel events for investigation, 622–623

DTrace, converting to SystemTap
 built-in variables, 667
 count syscalls by process name, 670–671
 count syscalls by syscall name, 671–672
 functionality, 665–666
 functions, 668
 listing syscall entry probes, 668
 overview of, 665
 probes, 666–667
 references, 674
 sample kernel stacks at 100 hz, 674
 summarize read() latency for "mysqld"
 processes, 672–673
 summarize read() returned size, 668–670
 terminology, 666
 trace file open(s) with process name/path name, 672
 trace new processes with process name/arguments, 673

DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and Free BSD (Gregg), 143

DTrace one-liners
 allocation tracing and, 310–311
 buffered tracing and, 177–179
 converting to SystemTap. *see* DTrace, converting to SystemTap
 fbt provider, 658–659
 io provider, 660
 ip provider, 661–662
 overview of, 141
 pid provider, 659–660
 proc provider, 655
 profile provider, 655–657
 sampling CPU usage, 237–238
 sched provider, 657
 syscall provider, 651–654
 sysinfo provider, 660–661
 tcp provider, 662

udp provider, 663
 vminfo provider, 661

dtruss
 buffered tracing, 177
 system-wide tracing, 118

Duplicate ACK detection, TCP, 485

Dynamic priority, scheduling classes and, 210

Dynamic random-access memory (DRAM)
 as common type of main memory, 273
 defined, 685

Dynamic sizing, in cloud capacity planning, 549

Dynamic ticks, 88

Dynamic tracing
 analyzing disks, 450–451
 analyzing file systems, 373–375
 applying to slow disk case study, 10
 defined, 685
 DTrace options, 134–135
 as performance metric, 7–8

Dynamic Tracing Guide, 143

E

ECC (error-correcting code)
 defined, 685
 magnetic rotational disks, 410

Elevator seeking
 algorithm, 397, 419–420, 457
 magnetic rotational disks and, 410

Encapsulation, 478

End-to-end arguments, for buffering, 481

Enterprise systems, cloud computing simulating, 57–58

Entire stack, defined in terms of systems performance, 1

EPT (extended page tables), 569

Erlang's C formula, 62

Error-correcting code (ECC)
 defined, 685
 magnetic rotational disks, 410

Errors
 analyzing, 495
 applying USE method to applications, 181–182
 applying USE method to CPUs, 216
 benchmarking, 593
 interpreting metrics, 48
 passive benchmarking and, 604
 performance monitoring of, 498
 USE method and, 42–43

etc (configuration files), in top-level directories, 100

- Ethereal, 520
 - Ethernet
 - defined, 685
 - physical network interfaces and, 487
 - Event tracing. *see also* Tracing
 - disk I/O events, 427–428, 442–444
 - file system events, 358–359
 - overview of, 53–54
 - SCSI events, 449–450
 - slow events, 372–373
 - TCP events, 525–526
 - Event-based concurrency, 160
 - Exceptions, kernel handling, 89
 - exec ()
 - function of key system calls, 95
 - in process execution, 93
 - execsnoop
 - buffered tracing, 177
 - system-wide tracing, 118
 - Expander card, 685
 - Experimentation
 - experimental analysis, 57
 - experimental tools, 535–536
 - quantifying performance issues, 70
 - testing CPU performance, 254–255
 - Experimentation, disk-related
 - ad hoc tests of sequential performance, 465
 - custom load generators, 465
 - micro-benchmarking tools, 466
 - random read example, 466–467
 - Experimentation, file system-related
 - ad hoc tests of sequential performance, 384
 - cache flushing, 387
 - micro-benchmarking tools, 384–386
 - overview of, 383–384
 - ext file systems
 - architecture of, 347–348
 - exposing ext4 file system internals, 371–372
 - tuning, 389
 - Extended page tables (EPT), 569
- F**
- FACK (forward acknowledgement) algorithm, 486, 539
 - Factor analysis
 - in capacity planning, 68
 - changing multiple benchmark factors, 594
 - Fair-share scheduler, CPU, 572
 - Fair-share scheduler (FSS), for CPU shares, 556
 - False sharing, hash tables and, 162
 - Fast recovery, TCP, 484
 - Fast retransmit, TCP, 484
 - fbt provider, DTrace
 - one-liners for, 658–659
 - tracing socket internals, 525
 - tracing TCP events, 525–526
 - fcachestat, printing cache activity on UFS, 379–380
 - FFS, 345–347
 - File descriptor
 - defined, 685
 - examining software resources, 48
 - File store, for persistent storage, 550
 - File system cache, 11, 271, 325–326
 - File system check. *see* fsck (file system check)
 - File systems
 - access timestamps, 336
 - ad hoc tests of sequential performance, 384
 - analysis tools, 362–363
 - analyzing (ksat), 381–382
 - analyzing file system cache with *vmstat*, 376–377
 - analyzing file system statistics with *sar*, 377–378
 - analyzing kernel memory usage on Solaris systems, 379
 - analyzing kernel slab caches related to file systems, 378–379
 - application calls for tuning, 387–388
 - architecture, 337
 - block device I/O stacks, 370–371
 - btrfs, 351
 - buffer cache, 339–340, 341
 - cache flushing, 387
 - cache tuning, 360
 - cache types, 339
 - caching, 327–328
 - capacity, 337
 - causes of slow disks, 11
 - checking capacity, 563
 - checking resource controls, 562
 - checking syscalls, 628–629
 - COW (copy-on-write) strategy, 344
 - debugging syscall interface, 364–365
 - dentry (Dcache) cache, 343
 - disk analysis, 353
 - DNLC (directory lookup cache), 341
 - DTrace analysis of, 365
 - dynamic tracing, 373–375
 - event tracing, 358–359
 - exercise solutions, 676
 - exercises and references, 391–393
 - experimentation, 383–384
 - ext, 347–348
 - features, 344–345

File systems (*continued*)

- FFS, 345–347
- file system cache, 11, 271, 325–326
- file system statistics (*fsstat*), 364
- free analysis, 376
- inode cache, 343
- interfaces, 325
- I/O stack model, 101, 337
- latency, 327, 354–356
- LatencyTOP tool, 375
- logical vs. physical I/O, 333–335
- measuring system latency with *DTrace*, 367–368
- measuring VFS latency with *DTrace*, 368–370
- memory summary using *mdb::memstat*, 380–381
- memory summary with `/proc/meminfo`, 380
- memory-based, 360–361
- memory-mapped files, 332–333
- metadata, 333
- methodologies, 353
- micro-benchmarking, 361–362, 384–386, 598–599
- non-blocking I/O, 332
- operation counts with *DTrace*, 365–366
- operation performance and, 335–336
- organization of files, 99
- other tools, 382–383
- overhead in OS virtualization, 555
- overview of, 99–100, 323
- page cache, 340–341, 342–343
- paging, 268
- performance monitoring, 358
- performance tuning targets, 22
- prefetch feature, 329–330
- printing cache activity on UFS using *fcachestat*, 379–380
- random vs. sequential I/O, 328
- raw and direct I/O, 331–332
- record size and performance trade-offs, 21
- resource controls, 557–558, 574
- second-level cache, 326
- special, 336
- static performance tuning, 359–360
- summarizing event counts with *DTrace*, 367
- synchronous writes, 331
- SystemTap analysis of, 375
- terminology regarding, 324
- top* analysis, 376
- tracing slow events, 372–373
- tracing ZFS read latency, 371–372
- tuning, 387, 389–391
- types of, 345

- UFS (Unix File System), 347
- VFS (virtual file system), 100–101, 337–338
- virtual file system statistics (*vfsstat*), 363–364
- visualizations, 383
- volumes and pools, 351–352
- workload characterization, 356–358
- workload separation, 360
- write-back caching, 330
- ZFS, 348–351

FileBench, 386

fio (Flexible IO Tester), 385–386

First-byte latency, 480, 524

Five Whys, in drill-down analysis, 51

Flame graphs, 253–254

Flash disks, 402

Flash-memory, 412

Flexible IO Tester (*fio*), 385–386

Floating-point operations per second (FLOPS), 601

FLOPS (floating-point operations per second), 601

fmddump, checking CPU with USE method, 623

fop_write() time, interrogating kernel, 630

fork() system call

- creating processes, 93
- function of, 95

Forward acknowledgement (FACK) algorithm, 486, 539

Fragmentation, as cause of slow disks, 11

Frames

- defined, 474
- physical network interfaces sending, 487

free command, showing memory and swap statistics, 376

Free lists

- defined, 278
- freeing memory, 280–281

FreeBSD jails, OS virtualization and, 552

Freeing memory

- free lists, 280–281
- overview of, 278–280
- page scanning for, 282–284
- reaping for, 281–282

Friendly fire issues, benchmarks, 595

Front-side bus, in Intel processors, 204–205

fsck (file system check)

- defined, 685
- journaling avoiding need for, 348
- Linux ext4 file system with faster, 348
- log replay avoiding need for, 347

FSS (fair-share scheduler), for CPU shares, 556

fsstat, for file system statistics, 364

`fsync()`, 630–633
 Full distribution latency, 498
 Full virtualization, in hardware virtualization, 555
 Full-duplex mode, 482
 Fully preemptable kernels, 104
 Functional block diagrams, in USE method, 45
 Functional unit, CPU processing instructions, 193–194
 Functions, DTrace to SystemTap, 668

G

Garbage collection, 166–167
`gbd`, in per-process tracing, 119
 Geometric mean, 70–71
 GLDv3 software, 491
`libc` allocator, 288
 Global zone, in OS virtualization, 551–552
 Google Compute Engine, 557
 Guests

- comparing virtualization technology complexity, 582
- exits mitigating hardware virtualization overhead, 567–569
- limits in OS virtualization, 558
- multiple server instances running as, 546
- multitenancy and, 551
- observability in hardware virtualization, 580
- observability in hardware virtualization for privileged guest/host, 576–577
- observability in OS virtualization, 561–562
- resource controls in OS virtualization, 555–558
- zones in OS virtualization, 551–552

 Gunther, Dr. Neil, 61

H

`halt` instruction, mitigating hardware virtualization overhead, 568–569
`handle_halt()`, mitigating hardware virtualization overhead, 568
 Hardware

- events, 206
- monitoring resource limits, 66
- thread, 190

 Hardware, CPU-related

- cache, 200–204
- CPCs (CPU performance counters), 206–208
- interconnects, 204–206
- MMU (memory management unit), 204
- overview of, 199
- processors, 199–200

Hardware, memory-related

- busses, 274–276
- main memory, 273–274
- MMU (memory management unit), 276–277
- overview of, 273

 Hardware, network-related

- controllers, 487
- interfaces, 487
- others, 488
- overview of, 486
- switches and routers, 487–488

 Hardware virtualization

- comparing virtualization technology performances, 581
- hardware-assisted virtualization, 566
- hypervisors and, 564–565
- implementations of, 565
- observability, 574–581
- OS virtualization vs., 552
- overhead, 566–571
- resource controls, 572–574
- types of, 563–564

 Harmonic mean, 70–71
 Hash collisions, 162
 Hash tables, 161–162
 HBAs (host bus adaptors). *see* Disk controllers
 HDD (hard disk drives). *see* Magnetic rotational disks
`hdparm`, for setting disk device tunables, 469
 Heap growth, 286
 Heat maps

- in CPU visualization, 251–252
- in file system visualization, 383
- latency heat maps, 463
- offset heat maps, 462–463
- subsecond-offset heat maps, 252–253
- utilization heat maps, 463–465
- for visualizations, 79–80

 Hertz (Hz), 685
 High-priority traffic, resource controls for, 502
 Histograms, for visualization, 73–74
 Historical statistics, archiving and reporting, 509–511
 Hit ratio, of cache, 30–31
 Horizontal scaling

- cloud architecture based on, 547–548
- scaling solutions, 69

 Host bus adaptors (HBAs). *see* Disk controllers
 Hosts

- comparing virtualization technology complexity, 582
- defined, 685
- guests of physical, 546

- Hosts (*continued*)
 - observability in hardware virtualization for privileged guest/host, 576–577
 - observability in OS virtualization, 559–560
 - switches providing communication path between, 487
 - Hot cache, 32
 - HT (HyperTransport) interconnect, 205
 - httpd processes, 672
 - Hubs, 487
 - Huge pages, Linux
 - MPSS (multiple page size support), 277
 - tuning multiple page sizes, 317–318
 - Hybrid virtualization, hardware virtualization, 556
 - HyperTransport (HT) interconnect, 205
 - Hypervisors
 - advanced observability of, 578–579
 - analyzing hardware virtualization overhead, 567
 - comparing complexity of virtualization technologies, 582
 - observability in hardware virtualization, 575–578
 - types of, 556–557
 - Hz (hertz), 685
- I**
- IaaS (infrastructure as a service), 546
 - ICMP
 - “can’t fragment” error, 479
 - ping latency in, 479–480
 - testing network connectivity, 514
 - Identification, in drill-down analysis, 50
 - Idle memory, 278
 - Idle threads
 - load balancing and, 210
 - prioritization of, 213
 - ifconfig
 - network analysis with, 494
 - overview of, 511
 - tuning network interface on Linux, 539
 - Ignoring errors, issues in benchmarking, 593
 - Ignoring perturbations, issues in benchmarking, 593–594
 - Ignoring variance, issues in benchmarking, 593
 - illumos, 685
 - Industry-standard benchmarks
 - overview of, 56
 - SPEC, 602
 - TPC, 601–602
 - understanding, 601
 - Infrastructure as a service (IaaS), 546
 - Inode cache, 343
 - inotify, in analyzing file systems, 382
 - Instructions, CPU. *see* CPU instructions
 - Instructions per cycle (IPC), CPU metric, 194–195
 - Intel VTune Amplifier XE profiling tool, 119
 - Interconnects, for multiprocessor architectures, 204–206
 - Interfaces
 - block device, 418–421
 - defined, 474
 - disk, 414
 - file system, 325
 - io provider (DTrace) for visibility of block device interface, 443–444
 - negotiation, 482
 - netstat statistics for, 504–509
 - physical network, 487
 - utilization, 487
 - Interpreted languages, 165–166
 - Inter-process communication (IPC)
 - connecting via IP to localhost with IP sockets, 483
 - defined, 685
 - interrupt coalescing mode, 10 GbE networking, 493
 - Interrupt latency, 92
 - Interrupt priority level (IPL), 92–93
 - Interrupt request (IRQ), 490, 685
 - Interrupt service routine, 92
 - Interrupt threads, 92
 - Interrupts
 - CPU cross calls, 103
 - defined, 86
 - DTrace analysis of, 241–242
 - IPL (interrupt priority level), 92–93
 - IRQ (interrupt request), 490, 685
 - overview of, 91–92
 - I/O (input/output). *see also* Disk I/O
 - analyzing block device I/O stacks, 370–371
 - analyzing sockets with DTrace, 523–524
 - comparing virtualization technologies, 581
 - CPU-bound vs. I/O bound, 99
 - defined, 396
 - I/O latency, 396, 420
 - I/O profiling, 180–181
 - I/O sizing, 158–159, 403–404
 - I/O stack, 101, 337
 - I/O throttling, 557
 - I/O wait as performance metric, 406
 - IOPS (I/O per second), 7
 - logical vs. physical I/O, 333–335

- mitigating hardware virtualization overhead, 570–571
 - non-blocking I/O, 162–163, 332
 - overhead in OS virtualization, 554–555
 - random vs. sequential I/O, 328
 - raw and direct I/O, 331–332
 - resource controls, 557–558, 574
 - resources studied as queueing systems, 45
 - slow disk case study, 9–11
 - io provider, DTrace
 - one-liners for, 660
 - for visibility of block device interface, 443–444
 - ioctl(), function of key system calls, 96
 - ionice, for scheduling and prioritizing processes, 468
 - IOPS (I/O per second)
 - calculating disk service time, 400
 - characterizing disk I/O workload, 424–425
 - defined, 16, 685
 - factors in comparing devices and workloads, 404
 - limitations as latency metric, 7
 - metrics of systems performance, 27
 - in micro-benchmarking disks, 430
 - purchasing rate for network storage, 550
 - read/write ratio and, 403
 - iosnoop
 - system-wide tracing, 118
 - tracing disk I/O with, 455–457
 - tracing storage device with, 53
 - iostat
 - checking disks, 625–626
 - summarizing per-disk I/O statistics, 432
 - summarizing per-disk I/O statistics on Linux, 433–436
 - summarizing per-disk I/O statistics on Solaris, 436–440
 - system-wide counters, 117
 - iostatop, top analysis of disks, 452–454
 - ip, 491, 511–512
 - IP header, 477
 - IP protocol, 478–479
 - ip provider, DTrace, 661–662
 - IP QoS (Quality of Service), resource controls for, 502
 - IP sockets, inter-process communication and, 483
 - IP stack, tuning in Solaris, 539–540
 - ipadm
 - replacing ifconfig on Solaris, 511
 - tuning with, 539–540
 - IPC (instructions per cycle), CPU metric, 194–195
 - IPC (inter-process communication)
 - connecting via IP to localhost with IP sockets, 483
 - defined, 685
 - iperf experimental tool, for network analysis, 535–536
 - IPL (interrupt priority level), 92–93
 - IRIX, 685
 - IRQ (interrupt request), 490, 685
 - irqbalancer process, 490
- J**
- JBOD (just a bunch of disks), 415
 - Jiffies, unit of time in Linux, 89
 - Joyent Cloud Analytics, 81, 383
 - Joyent public cloud
 - OS virtualization, 552
 - resource controls in OS virtualization, 555–558
 - using KVM hardware virtualization, 557, 565
 - Joyent SmartOS, 553
 - Jumbo frames
 - for packet size, 478
 - tuning network performance with, 542
 - Just a bunch of disks (JBOD), 415
- K**
- Keep-alive strategy, for network connections, 481
 - Kendall's notation, in categorizing factors of queueing systems, 63–64
 - Kernel
 - analyzing memory usage, 379
 - clock, 88–89
 - comparing Solaris and Linux, 112–113
 - defined, 86
 - DTrace to SystemTap conversion example, 674
 - execution, 88
 - file system support, 99
 - interrogating in Redis case study, 629–631
 - Linux-based, 109–112
 - mode, 88–89, 154, 196
 - overview of, 87
 - preemption, 103–104
 - profiling, 236–237
 - schedulers, 98–99
 - selecting events for investigation, 622
 - Solaris-based, 106–109
 - stacks, 90–91, 674
 - tracing, 131
 - Unix kernel, 106
 - versions, 105

- Kernel level, 685
 - Kernel mode
 - analysis of time in, 196
 - determining CPU mode of application, 154
 - overview of, 88–89
 - Kernel-space, 86
 - kmastat, analyzing kernel memory usage on
 - Solaris systems, 302–303, 379
 - Knee point
 - modeling scalability, 57
 - in scalability, 24–25
 - visualizing scalability profiles, 60
 - Known workloads, verifying observations, 150
 - Known-knowns, in performance analysis, 26
 - Known-unknowns, in performance analysis, 26
 - kstat (kernel statistics)
 - analyzing file systems, 381–382
 - checking CPU, 623
 - overview of, 127–130
 - kswapd, page-out daemon in Linux, 282–283
 - KVM
 - advanced observability in, 578–579
 - observability in hardware virtualization for
 - privileged guest/host, 575–577
 - kvm_emulate_halt(), mitigating hardware
 - virtualization overhead, 569
- L**
- Language virtual machines, 166
 - Large segment offload, improving performance of
 - MTU frames, 479
 - Latency
 - analysis in general, 51–52
 - application performance objectives, 155
 - vs. completion time, 78–80
 - connection latency, 480–481
 - CPU cache latency, 202
 - defined, 16, 474
 - disk analysis, 426–427
 - disk I/O latency, 399–401
 - distribution of, 73
 - DTrace to SystemTap example, 672–673
 - in expressing application performance, 35
 - in extreme environments, 23
 - file system analysis, 354–356
 - file system latency, 327
 - first-byte latency, 480
 - heat maps of, 463
 - interrupt latency, 92
 - I/O latency policies, 420
 - line chart visualization of, 77
 - main memory and, 273
 - measuring, 27, 479
 - name resolution and, 479
 - network analysis and, 497–498
 - network storage access increasing, 550
 - overview of, 18–19
 - perf for checking scheduler latency, 245–246
 - as performance metric, 6–7
 - ping latency, 479–480
 - ramping load and measuring, 610–611
 - reporting on file system latency, 375
 - round-trip time and, 481
 - scheduler latency, 192
 - in slow disk case study, 10
 - socket analysis, 524–525
 - summarizing disk latency with DTrace,
 - 446–447
 - targets for workload analysis, 34
 - tick latency, 88
 - time scale of, 19–20
 - Latency, in Redis application
 - causes of read latency, 626–627
 - measuring file system syscalls, 628–629
 - problem statement, 618–619
 - Latency outliers
 - baseline statistics and, 54
 - defined, 396
 - garbage collection and, 167
 - LatencyTOP tool, 375
 - Layer terminology, of protocol stack, 476
 - Leak detection, 293–294
 - Least recently used (LRU)
 - cache management algorithm, 31
 - defined, 686
 - LFU (least frequently used), cache management
 - algorithm, 31
 - lgrps (locality groups), Solaris, 214
 - libc allocator, 288
 - libumem slab allocator, 288, 309–310
 - Limits, in OS virtualization, 555, 558
 - Line charts
 - applied to disks, 461
 - for visualizations, 77–78
 - Linear scalability, 59
 - Link aggregation, tuning network performance
 - with, 542
 - Linux
 - analyzing file system cache, 376
 - analyzing file system statistics, 377–378
 - block device interface and, 419–420
 - breakpoint tracing on, 173–175
 - comparing Solaris and Linux kernels,
 - 112–113
 - configuring network interfaces, 511

- CPU performance tools, 250
- debugging syscall interface, 364–365
- delay accounting, 130–131
- flushing system cache, 387
- free lists, 281
- freeing memory, 279
- history of Linux kernel, 109–110
- huge pages, 277, 317–318
- interface statistics, 504–506
- investigating library calls, 182
- kernel versions and syscall counts, 105
- load averages in uptime analysis, 226
- lxc containers in OS virtualization, 552
- mapping process memory, 307–308
- measuring VFS latency with DTrace, 369–370
- memory management, 98
- memory performance tools, 312–313
- memory summary with `/proc/meminfo`, 380
- monitoring, 120
- network analysis tools, 503
- network stack, 489–490
- observability sources, 121
- other network performance tools, 534
- overcommit supported by, 270
- page scanning methods, 282–283
- performance features of Linux kernel, 110–112
- `/proc` file system, 122–124
- process binding, 259
- process status (`ps`), 304
- processor sets, 222
- profiling tools, 119
- resource control groups (`cgroups`), 104
- resource controls, 260, 318
- scheduler options, 257
- scheduling and prioritizing processes (`ionice`), 468
- scheduling classes, 211–212
- scheduling domains, 214
- scheduling priority and class, 256
- setting disk device tunables (`hdparm`), 469
- slab cache usage (`slabtop`), 302
- summarizing per-disk I/O statistics (`iostat`), 433–436
- `/sys` file system, 126–127
- system activity reporters (`sar`), 299–300, 440–441, 509–511, 649–650
- thread-state analysis in, 170
- time sharing on, 210
- top analysis of disk I/O (`iostat`), 452–453
- tracing block device events on (`blktrace`), 457–459
- tuning, 536–539
 - tuning memory, 315–316
 - tuning multiple page sizes, 317
 - USE method checklist for physical resources, 637–641
 - USE method checklist for software resources, 641–642
 - virtual memory statistics (`vmstat`), 296–297
 - voluntary kernel preemption in, 104
- Linux Performance Events (LPE). *see* `perf`
- Listen backlog, TCP, 492
- Little's Law, 62
- Live patching, of kernel address space, 135
- LNU (not frequently used), cache management algorithm, 32
- Load. *see* Workload
- Load averages, in analyzing CPUs, 224–226
- Load balancing
 - cloud architecture, 547
 - idle threads and, 210
 - key functions of CPU scheduler, 209
 - scaling solutions and, 69
- Load generators
 - in disk-related experimentation, 465
 - micro-benchmarking with, 56
- Local connections
 - network, 482–483
 - networks, 482–483
- Local disks, 686
- Local memory, 274
- Localhost
 - connections, 482
 - defined, 477
- Lock analysis, analyzing and tuning applications, 182–185
- `lockstat`, 183, 185
- Logical CPU, 190
- Logical I/O, 333–335
- Logical processor, 686
- Logs/logging
 - applications, 154
 - packet capture logs, 498–500
 - system logs, 118
- Loopback, localhost connections using, 482
- Low-priority traffic, resource controls for, 502
- LPE (Linux Performance Events). *see* `perf`
- LRU (least recently used)
 - cache management algorithm, 31
 - defined, 686
- LSI MegaCli
 - analyzing disk controllers, 459–460
 - tuning disk controllers, 469–470
- `ltrace`, investigating library calls on Linux, 182
- lx Branded Zones, 552

lxc Linux Containers, in OS virtualization, 552
 lxproc, Linux-like proc on Solaris, 126

M

Macro-benchmarks, 599. *see also* Simulations
 madvise(), application calls, 388

Magnetic rotational disks
 defined, 685
 overview of, 408
 types of disks, 395

Main board (or system board), 686

Main memory
 architecture, 273–274
 busses, 274–276
 as capacity resources, 45
 checking resource controls, 562
 defined, 266
 DRAM, 273
 oversubscription of, 267
 utilization and saturation, 271–272

Maintenance, comparing virtualization technologies for, 583

Major fault, 686

malloc, in allocation tracing, 308

Marketing, benchmarking during, 588

Markov model, of stateful workload simulation, 600

Markovian arrivals, (M/D/1, M/M/1, M/M/c, and M/G/1), 64–65

Maximum transmission unit (MTU)
 improving frame performance, 479
 limiting packet size, 478

mdb, Solaris

 analyzing kernel memory usage (kmastat), 302–303, 379
 per-process tracing, 119
 setting tunable parameters, 539
 summarizing memory usage (memstat), 380–381

Mean, in statistics, 70–71

Medians

 plotting, 77–78
 in statistics, 72

MegaCli

 analyzing disk controllers, 459–460
 tuning disk controllers, 469–470

meminfo, memory summary with /proc/
 meminfo, 380

Memory

 allocators, 272, 286–289
 analysis tools, 295
 architecture, 272–273

 busses, 274–276

 characterizing usage of, 291–292

 checking resource controls, 562

 comparing virtualization technology
 performance overhead, 581–582

 CPU memory cache, 191–192

 cycle analysis, 293

 demand paging, 269–270

 DTrace for allocation tracing, 308–311

 DTrace for fault tracing, 311–312

 DTrace for tracing page-out daemon, 312

 exercise solutions, 676

 exercises and references, 319–321

 file system cache and, 271

 free lists, 280–281

 freeing, 278–280

 growth, 166–167

 hardware, 273

 heap growth and, 286

 improving performance of multithreaded
 applications, 318

 kernel memory on Solaris systems (kmastat), 302–303

 leak detection, 293–294

 main memory, 273–274

 managing, 97–98

 mapping process memory (pmap), 306–308

 methodologies, 289

 micro-benchmarking, 294–295

 micro-benchmarking to check memory access,
 223

 mitigating hardware virtualization overhead,
 569–570

 MMU (memory management unit), 276–277

 overcommit support in Linux, 270

 overview of, 265–266

 page scanning for freeing, 282–284

 paging, 268–269

 performance monitoring, 293

 performance tools for analyzing, 312–314

 performance trade-offs between CPU and, 21

 primary, secondary, and virtual, 97

 process address space and, 284–285

 process status (ps), 304–305

 reaping for freeing memory, 281–282

 regulating access to, 161

 resource controls, 294

 resource controls in hardware virtualization,
 573–574

 resource controls in OS virtualization, 556

 resource controls in tuning, 318

 showing statistics with free, 376

 slab cache usage (slabtop), 302

- software, 278
- static performance tuning, 294
- swapping, 271
- system activity reporters (sar), 298–301
- SystemTap, 312
- terminology regarding, 266
- tools method, 289–290
- top running processes, 305–306
- tunable parameters, 314–317
- tuning multiple page sizes, 317–318
- USE method for improving performance, 290–291
- utilization and saturation, 271–272
- virtual memory, 267–268
- virtual memory statistics (vmstat), 295–298
- word size and, 272
- Memory, in Redis application case study
 - checking for memory errors, 624–625
 - conclusions regarding, 626–627
 - epilogue, 633
- Memory locality, NUMA system, 192
- Memory management unit. *see* MMU (memory management unit)
- Memory nodes, 274
- Memory-based file systems, 360–361
- Memory-mapped files, 332–333
- memstat, memory summary using
 - mdb::memstat, 380–381
- Metadata, 333
- Method R, in performance analysis, 52
- Methodologies
 - ad hoc checklists, 38–39
 - anti-methods, 36–38
 - baseline statistics, 54
 - cache tuning, 55–56
 - capacity planning, 65–69
 - categorizing types of, 35–36
 - cloud computing and, 48
 - diagnosis cycle, 41
 - drill-down analysis, 50–51
 - event tracing, 53–54
 - exercise solutions, 675
 - exercises and references, 82–83
 - Five Whys, 51
 - functional block diagrams in USE method, 45
 - interpreting metrics, 48
 - latency analysis, 51–52
 - Method R, 52
 - metrics of USE method, 43–47
 - micro-benchmarking, 56–57
 - modeling, 17–18, 57–64
 - monitoring, 74–76
 - overview of, 15
 - problem statement, 39
 - resource lists in USE method, 44–45
 - scientific method, 39–41
 - software resource examination, 47–48
 - static performance tuning, 55
 - statistics, 69–74
 - tools method, 41–42
 - USE method, 42–43
 - visualizations, 76–81
 - workload characterization, 49–50
- Methodologies, application-related
 - breakpoint tracing, 173–177
 - buffered tracing, 177–179
 - CPU profiling, 171–173
 - drill-down analysis applied to, 182
 - I/O profiling, 180–181
 - lock analysis applied to, 182–185
 - overview of, 167–168
 - static performance tuning, 185–186
 - syscall analysis, 173
 - thread-state analysis, 168–171
 - USE method, 181–182
 - workload characterization, 181
- Methodologies, benchmark-related
 - active benchmarking, 604–606
 - CPU profiling, 606–607
 - custom benchmarks, 608
 - overview of, 602–603
 - passive benchmarking, 603–604
 - ramping load, 608–611
 - sanity check, 611
 - statistical analysis, 612–613
 - USE method, 607–608
 - workload characterization, 608
- Methodologies, CPU-related
 - CPU binding, 222
 - cycle analysis, 219–220
 - micro-benchmarking, 222–223
 - overview of, 214–215
 - performance monitoring, 220
 - priority tuning, 221–222
 - profiling, 218–219
 - resource controls, 222
 - scaling, 223–224
 - static performance tuning, 220–221
 - tools method, 215
 - USE method, 216
 - workload characterization, 216–218
- Methodologies, disk-related
 - cache tuning, 429
 - event tracing, 427–428
 - latency analysis, 426–427
 - micro-benchmarking, 429–430

- Methodologies, disk-related (*continued*)
 - overview of, 421
 - performance monitoring, 423–424
 - resource controls, 429
 - scaling solutions, 431
 - static performance tuning, 428–429
 - tools method, 422
 - USE method, 422–423
 - workload characterization, 424–426
- Methodologies, file system-related
 - cache tuning, 360
 - disk analysis, 353
 - event tracing, 358–359
 - latency analysis, 354–356
 - memory-based, 360–361
 - micro-benchmarking, 361–362
 - overview of, 353
 - performance monitoring, 358
 - static performance tuning, 359–360
 - workload characterization, 356–358
 - workload separation, 360
- Methodologies, memory-related
 - characterizing memory usage, 291–292
 - cycle analysis, 293
 - leak detection, 293–294
 - micro-benchmarking, 294–295
 - overview of, 289
 - performance monitoring, 293
 - resource controls, 294
 - static performance tuning, 294
 - tools method, 289–290
 - USE method, 290–291
- Methodologies, network performance
 - drill-down analysis, 500–501
 - latency analysis, 497–498
 - micro-benchmarking, 502–503
 - packet sniffing, 498–500
 - performance monitoring, 498
 - resource controls, 501
 - static performance tuning, 501
 - TCP analysis, 500
 - tools method, 494–495
 - USE method, 495–496
 - workload characterization, 496–497
- Metrics. *see also* Statistics
 - application, 154
 - CPI (cycles per instruction), 194–195
 - disk I/O, 423–424
 - dynamic tracing and, 7–8
 - latency as, 6–7
 - quantifying performance issues, 6
 - for resource analysis, 33
 - types of, 27
 - of USE method, 43–48
 - verifying observations, 150–151
 - for workload analysis, 35
- MFU (most frequently used), cache management
 - algorithm, 31
- M/G/1 Markovian arrivals, 64
- Micro-benchmarking
 - applying to CPUs, 222–223
 - applying to disks, 429–430
 - applying to file systems, 361–362
 - comparing virtualization technologies with, 582
 - design example, 598–599
 - in determining speed of main memory, 294–295
 - in disk-related experimentation, 466
 - network analysis with, 502–503, 535
 - overview of, 56–57
 - tools, 384–386
 - understanding, 597–598
 - variance in, 593
- Microprocessors, 488. *see also* CPUs (central processing units); Processors
- Microstate accounting, Solaris, 131, 170–171
- Millions of instructions per second (MIPS),
 - industry benchmark metric, 601
- Minor fault, 686
- MIPS (millions of instructions per second),
 - industry benchmark metric, 601
- Mirroring, RAID-1, 416
- Misleading issue, in benchmarking, 595–596
- M/M/1 Markovian arrivals, 64
- M/M/c Markovian arrivals, 64
- MMU (memory management unit)
 - mitigating hardware virtualization memory mapping overhead, 569
 - overview of, 276–277
 - virtual to physical address translation, 204
- Models, network-related
 - controller, 475
 - network interface, 474–475
 - protocol stack, 476
- Models/modeling
 - creating theoretical model of Redis
 - application, 623
 - first steps in performance engineering, 2
 - I/O stack model, 101
 - Markov model in workload simulation, 600
 - non-blocking I/O model, 162–163
 - overview of, 57
 - queuing model, 17–18
 - queuing theory, 61–65
 - scalability and, 60–62

- simulating enterprise systems using cloud
 - computing, 57–58
 - sparse-root zones model, 555
 - SUT (system under test), 17
 - visual identification, 58–60
 - wireframe models, 80
 - Model-specific registers (MSR), 207
 - Monitoring
 - baseline statistics and, 54
 - CPU performance, 220
 - disk performance, 423–424
 - in drill-down analysis, 50
 - events, 206
 - file system performance, 358
 - key metrics for, 498
 - memory metrics, 293
 - network analysis with, 498
 - overview of, 74
 - software in OS virtualization, 555
 - summary-since-boot values and, 76
 - third-party products for, 76
 - time-based patterns, 74–76
 - tools, 120
 - Most frequently used (MFU), cache management algorithm, 31
 - Most recently used (MRU), 31
 - mount, analyzing file systems, 382
 - Mount points, directory, 99
 - Mounting file system, to directory, 99
 - MPSS (Multiple page size support), 277
 - mpstat
 - analyzing CPUs, 227–229
 - checking for CPU errors in Redis application, 623
 - observability in OS virtualization, 561
 - system-wide counters, 117
 - MR-IOV (Multi Root I/O Virtualization), 570–571
 - MRU (most recently used), 31
 - MSR (model-specific registers), 207
 - mtmalloc allocator, 288
 - MTU (maximum transmission unit)
 - improving frame performance, 479
 - limiting packet size, 478
 - Multi Root I/O Virtualization (MR-IOV), 570–571
 - Multicast transmission, 477
 - Multichannel architecture, in memory buses, 275–276
 - Multimodal distributions, in statistics, 73–74
 - Multiple page size support (MPSS), 277
 - Multiprocessors. *see also* CPUs (central processing units)
 - CPU scalability and, 197–198
 - interconnects for, 204–206
 - multiprocessor statistics (mpstat), 227–229
 - parallelism and, 160
 - support for, 103
 - Multitenancy, 546, 550–551
 - Multithreading
 - allocators for improving performance, 318
 - CPU scalability and, 197–198
 - lock analysis applied to, 182–183
 - overview of, 160
 - scaling hardware, 190
 - Mutex (MUTually EXclusive) locks
 - examining software resources, 47
 - types of synchronization primitives, 161
 - mysqld, 672–673, 686
- ## N
- Nagle algorithm, for TCP congestion control, 485–486
 - name resolution latency, 479
 - NAS (network-attached storage), 417–418
 - Native (bare-metal) hypervisor, 556
 - ndd, Solaris, 539
 - Nested page tables (NPT), 569
 - netstat
 - checking Redis application for network errors, 626
 - interface statistics on Linux, 504–506
 - interface statistics on Solaris, 507–509
 - investigating Redis application performance issues, 620–622
 - network analysis with, 494
 - overview of, 503–504
 - studying statistics for areas to investigate, 622
 - system-wide counters, 117
 - Network devices
 - drivers, 493
 - tuning on Solaris, 541
 - Network interface, 102
 - checking Redis application for network errors, 626
 - negotiation, 482
 - overview of, 474–475
 - tuning on Linux, 539
 - utilization, 482
 - workload characterization of, 496
 - Network interface card. *see* NIC (network interface card)
 - Network I/O, 563. *see also* Disk I/O
 - Network latency, 497
 - Network packets. *see* Packets
 - Network protocols, 102

- Network stack
 - Linux, 489–490
 - overview of, 488–489
 - Solaris, 491
- Network-attached storage (NAS), 417–418
- Networks/networking
 - advanced network tracking, 531–533
 - analysis, 503
 - buffering, 481
 - capturing and inspecting packets (snoop), 517–520
 - capturing and inspecting packets (tcpdump), 516–517
 - configuring network interfaces, 511
 - configuring network interfaces and routes (ip), 512
 - connection backlog, 481
 - drill-down analysis, 500–501
 - DTrace analysis of backlog drops, 529–531
 - DTrace analysis of packet transmission, 527–529
 - DTrace analysis of sockets, 521–525
 - DTrace analysis of TCP events, 525–526
 - DTrace network providers, 520–521
 - encapsulation, 478
 - exercises and references, 542–544
 - experimental tools, 535–536
 - hardware, 486–488
 - interface negotiation, 482
 - latency, 479–481
 - latency analysis, 497–498
 - local connections, 482–483
 - methodologies, 493–494
 - micro-benchmarking, 502–503
 - models, 474–476
 - other performance tools, 534
 - overview of, 102
 - packet size, 478–479
 - packet sniffing, 498–500
 - perf for static and dynamic event tracing, 533–534
 - performance monitoring, 498
 - printing network interface statistics (nicstat), 512–513
 - protocols, 477–478, 483–486
 - providing interface statistics on Solaris (dladm), 513–514
 - reporting on network activity (sar), 509–511
 - reporting on network statistics (netstat), 503–509
 - resource controls, 502
 - routing, 476–477
 - software, 488–493
 - static performance tuning, 501
 - statistics, 509–511
 - SystemTap analysis of file system events, 533
 - TCP analysis, 500
 - terminology, 474
 - testing network connectivity (ping), 514
 - testing routes and bandwidth between routes (pathchar), 515–516
 - testing routes (traceroute), 514–515
 - tools method, 494–495
 - tuning Linux, 536–539
 - tuning Solaris, 539–542
 - USE method, 495–496
 - utilization, 482
 - Wireshark tool for packet capture and analysis, 520
 - workload characterization, 496–497
- NFS, 100
- NIC (network interface card)
 - housing network controller, 475
 - improving performance of MTU frames, 479
 - overview of, 102
- nice, for scheduling priority and class, 256–257
- nicstat
 - checking Redis application for network errors, 626
 - network analysis tool, 512–513
- nmap(), function of key system calls, 96
- Nodes, 274
- Noisy neighbors, 551
- Non-blocking I/O, 162–163, 332
- Non-idle time, utilization and, 29
- Non-regression testing, testing hardware or software changes, 11
- Non-uniform memory access. *see* NUMA (non-uniform memory access)
- Noop (no-operation) policy, I/O latency, 420
- Normal distribution, 72
- Not frequently used (LNU), cache management algorithm, 32
- NPT (nested page tables), 569
- NUMA (non-uniform memory access)
 - groups for scheduling and memory placement, 214
 - interconnects and, 204–206
 - main memory architecture, 273–274
 - memory locality and, 192
- Numbers without analysis, benchmark issues, 591–592

O

- Object store, persistent storage, 550
- Objectives, application performance
 - Big O notation applied to performance analysis, 156–158
 - optimizing common code path, 156
 - overview of, 155–156
 - role of observability in eliminating unnecessary work, 156
- Objectives, in performance engineering, 2
- Observability
 - comparing virtualization technology performances, 582–583
 - eliminating unnecessary work in applications, 156
 - in hardware virtualization, 574–581
 - in OS virtualization, 558–563
 - OSs (operating systems) and, 104
 - in quantifying performance issues, 69–70
 - of RAID disks, 415–416
- Observability tools
 - baseline statistics and, 54
 - counters, 116–117
 - DTrace tool. *see* DTrace
 - exercises and references, 151
 - identification process in drill-down analysis, 50
 - Linux delay accounting, 130–131
 - Linux /proc files, 122–124
 - Linux /sys file system, 126–127
 - monitoring tools, 120
 - overview of, 115
 - perf, 149
 - /proc file system, 121
 - profiling tools, 119–120
 - Solaris kstat files, 127–130
 - Solaris microstate accounting, 131
 - Solaris /proc files, 124–126
 - sources of performance statistics, 120–121, 131–133
 - SystemTap. *see* SystemTap
 - tracing tools, 118–119
 - types of, 116
 - verifying observations, 150–151
- Offset heat maps, 462–463
- On-disk cache, magnetic rotational disks, 410
- One-liners
 - DTrace. *see* DTrace one-liners
 - SystemTap, 146–148
- OOM (out of memory), 266
- OOM (out of memory) killer
 - freeing memory, 279
 - searching for out of memory, 290
- open(), function of key system calls, 95
- Open VZ (Open Virtuozzo), 552
- opensnoop
 - analyzing file systems, 367
 - buffered tracing, 177
- Operation counts, applying DTrace to file systems, 365–366
- Operation performance, in file systems, 335–336
- Operation rate
 - defined, 686
 - throughput and, 16
- Operation time, latency and, 18
- opprofile profiling tool, 119
- Optimizing, for the benchmark, 596–597
- Oracle Solaris Studio profiling tools, 119
- OS virtualization
 - comparing virtualization technologies, 581
 - observability and, 558–563
 - overhead of, 553–555
 - overview of, 48
 - partitioning in, 551–553
 - resource controls, 555–558
- OSI model, 476
- OSs (operating systems)
 - caching, 101–102
 - comparing Solaris and Linux kernels, 112–113
 - device drivers, 103
 - disk I/O stack, 418–421
 - exercise solutions, 675
 - exercises and references, 113–114
 - file systems, 99–101
 - interrupts and interrupt threads, 91–92
 - IPL (interrupt priority level), 92–93
 - kernel, 87–89, 105
 - kernel preemption, 103–104
 - Linux-based kernels, 109–112
 - memory management, 97–98
 - micro-benchmarking operations, 223
 - multiprocessor support, 103
 - networking and, 102
 - observability and, 104
 - overview of, 85
 - processes, 93–95
 - resource management, 104
 - schedulers, 98–99
 - Solaris-based kernels, 106–109
 - stacks, 89–91
 - system calls, 95–96
 - terminology regarding, 86
 - tuning, 467–469
 - Unix kernel, 106
 - virtual memory, 97

- Out of memory (OOM), 266
 - Out of memory (OOM) killer
 - freeing memory, 279
 - searching for out of memory, 290
 - Outliers
 - in heat maps, 79
 - latency outliers, 54, 167
 - in statistics, 74
 - Overcommit
 - defined, 268
 - Linux supported, 270
 - Overhead
 - comparing virtualization technologies, 581
 - DTrace, 143
 - hardware virtualization, 566–571
 - OS virtualization, 553–555
 - of performance metrics, 27
 - SystemTap, 148
 - tick overhead, 88
 - Overprovisioning, dynamic sizing in cloud and, 549
 - Oversubscription, of main memory, 267
- P**
- Packet drops, in Redis application
 - investigating, 620–622
 - Joyent analysis of, 619–620
 - in problem statement, 618–619
 - Packets
 - capture and inspection with snoop, 517–520
 - capture and inspection with tcpdump, 516–517
 - capture logs, 498–500
 - communicating by transferring, 478
 - defined, 474
 - event tracing, 53
 - managing, 477
 - monitoring out-of-order, 498
 - reducing overhead of, 478
 - round-trip time of, 481
 - router role in delivery, 488
 - size of, 478–479
 - sniffing, 115, 132, 498–500
 - tracing transmission, 527–528
 - Padding hash locks, 162
 - Page cache, 279, 340–343
 - Page faults
 - defined, 266
 - page mapping and, 269–270
 - tracing, 311–312
 - Page scanning
 - checking with tools method, 290
 - for freeing memory, 282–284
 - Page tables, reading, 204
 - Page-out daemon
 - page scanning methods, 282–283
 - tracing, 312
 - Pages
 - as memory unit, 266
 - MPSS (multiple page size support), 277
 - tuning multiple page sizes, 317–318
 - Paging
 - anonymous paging, 268–269
 - defined, 266
 - demand paging, 269–270
 - file system paging, 268
 - moving small units of memory, 97–98, 267
 - Solaris methods for freeing memory, 279
 - thread-state analysis and, 170
 - tools method for checking, 290
 - PAPI (Processor Application Programmers Interface), 207
 - Parallelism
 - application performance and, 160–162
 - defined, 686
 - lock analysis applied to, 182–183
 - Paravirtualization, hardware virtualization
 - defined, 566
 - mitigating I/O overhead, 570–571
 - overview of, 555–556
 - Passive benchmarking, 603–605
 - Passive TCP connection rate, 484, 496
 - pathchar, for network analysis, 515–516
 - pbind, process binding in Solaris, 259
 - PC (program counter), 686
 - PCI pass-through, 570–571
 - PDP (Programmed Data Processor), 686
 - Percentiles, in statistics, 72, 77–78
 - perf
 - advanced observability with, 577–578
 - analysis phase of drill-down analysis, 51
 - analyzing CPUs, 243–244
 - analyzing disks, 451–452
 - analyzing networks, 495
 - block trace points in disk analysis, 451–452
 - checking scheduler latency, 245–246
 - documentation, 249
 - drill-down analysis, 182
 - overview of, 149, 243, 533–534
 - profiling with, 119, 244–245
 - subcommands, 243
 - summarizing CPU cycle behavior (`stat`), 246–247
 - system-wide tracing with, 118
 - tools method for checking memory, 290
 - tracing software, 247–249

- Performance engineers, 2
- Performance instrumentation counters (PICs), 206, 249–250
- Performance isolation
 - cloud computing and, 9
 - resource controls in cloud computing for, 551
- Performance monitoring units (PMU), 206
- Per-interval average latency, 498
- Per-operation latency, 498
- Per-process observability tools
 - counters, 117
 - Linux /proc file system, 122–123
 - overview of, 116
 - profiling, 119
 - Solaris /proc files, 124–126
 - tracing, 118–119, 131
- Perspectives
 - overview of, 32
 - performance analysis, 4
 - resource analysis, 33–34
 - workload analysis, 34–35
- Perturbations, benchmarking, 593–594
- Physical I/O, 333–335
- Physical resources
 - USE method checklist in Linux, 637–640
 - USE method checklist in Solaris, 643–645
- PICs (performance instrumentation counters), 206, 249–250
- PID (process ID), 93–94
- pid provider, DTrace, 659–660
- pidstat
 - analyzing CPU usage by process or thread, 234
 - printing disk I/O statistics, 441–442
- ping, analyzing network connectivity, 514
- Ping latency, 479–480
- Pluggable device drivers, 103
- pmap (mapping process memory), 117, 306–308
- PMU (performance monitoring units), 206
- Point-in-Time recommendations, for
 - performance, 23
- Policies
 - I/O latency, 420
 - scheduling, 210, 212
- poll() syscall, 159–160
- pollsys(), 628, 633
- Pools
 - file systems, 351–352
 - observing ZFS pool statistics, 382–383
- POSIX (Portable Operating System Interface)
 - standard, 99, 686
- posix_fadvise(), application calls, 387–388
- Preemption
 - high priority threads and, 196
- kernel, 103–104
 - key functions of CPU scheduler, 209
 - triggering in Linux and Solaris, 210
- Prefetch feature, file systems, 329–330
- Price/performance ratio
 - cloud computing and, 546
 - comparing benchmarking systems, 589
 - industry-standard benchmarks measuring, 601
- printf(), 670–672
- Prioritization
 - inheritance in Solaris, 197
 - resource controls in hardware virtualization, 572–573
 - resource controls in OS virtualization, 555
 - scheduling classes, 210–211, 256–257
 - scheduling processes, 256–257
 - threads and priority inversion, 196–197
 - tuning priorities, 221–222
- Privileged guest/host, observability in hardware
 - virtualization, 575–578
- probefunc(), 671–672
- Probes
 - CPU scheduler, 242
 - DTrace, 135–136
 - DTrace to SystemTap, 666–667
 - SystemTap, 145
- Problem statement
 - overview of, 39
 - for Redis application case study, 618–619
- /proc file system
 - Linux, 122–124
 - Solaris, 124–126
 - sources of performance statistics, 121
- proc provider, DTrace, 655
- Process address space, 284–285
- Process binding, 222, 259
- Process ID (PID), 93–94
- Process virtual machines, 166
- Processes
 - creating, 93
 - defined, 86, 686
 - DTrace to SystemTap examples, 671–673
 - environment, 94–95
 - life cycle of, 94
 - monitoring. *see* ps (process status)
 - moving between main and secondary memory, 97–98
 - multiple, 160
 - process accounting, 132
 - running in user mode, 89
 - scheduling, 98–99, 256–257
 - viewing top running. *see* top (top running processes on Linux)

Processor Application Programmers Interface (PAPI), 207

Processor ring, 686

Processors. *see also* CPUs (central processing units)

binding, 163

CPUs and, 199–200

defined, 86, 190

microprocessors, 488

MSR (model-specific registers), 207

multiprocessor support, 103

multiprocessors. *see* Multiprocessors

options, 260

parallelism and, 160

processor groups, 214

processor sets, 222

virtual processor, 190

word size, 198–199

Process/thread capacity, examining software resources, 48

/proc/meminfo, providing summary of memory, 380

procsystime, for buffered tracing, 177

profile provider, DTrace, 655–657

Profiling

CPU profiling, 171–173, 218–219

DTrace applied to kernel profiling, 236–237

I/O profiling, 180–181

perf for process profiling, 245

perf for system profiling, 244–245

sampling and, 30

tools, 119–120

types of observability tools, 116

user profiling, 238–240

visualization of scalability profiles, 59

Program counter (PC), 686

Programmed Data Processor (PDP), 686

Programming languages

compiled languages, 164–165

garbage collection, 166–167

interpreted languages, 165–166

micro-benchmarking higher-level languages, 223

overview of, 163–164

virtual machines and, 166

Proof of concepts, benchmarking and, 588

Protocols, network

overview of, 102

performance characteristics of, 477–478

performance features of TCP, 483–484

protocol stack, 476

Providers, DTrace, 135–136

Providers, IaaS, 546

prstat (top running processes in Solaris)

analyzing CPUs, 232–234

analyzing memory usage, 290

checking Redis application for memory errors, 624

monitoring top running processes, 305–306

observability in OS virtualization, 562

ps (process status)

analyzing CPUs, 230–231

analyzing memory usage, 304–305

per-process counters, 117

psrset, creating CPU sets in Solaris, 259

ptime, reporting on CPU usage, 235–236

public cloud providers, 546

Q

QEMU (Quick Emulator), KVM

advanced observability with, 578–579

observability in hardware virtualization for privileged guest/host, 576

overview of, 557

QPI (Quick Path Interconnect), 205–206

Quantification, of performance issues, 6, 69–70

quantize() action, DTrace, 528

Queries, latency analysis applied to, 51–52

Queueing model, 17–18

Queueing networks, 61

Queueing systems

applied to M/D/1, 64–65

commonly studied, 64

I/O resources studied as, 45

Kendall's notation applied to categorizing factors of, 63

modeling hardware and software components, 61–62

Queueing theory

creating theoretical model of Redis

application, 623

overview of, 61–65

statistical analysis of, 613

Queues, TCP backlog, 492

Quick Emulator (QEMU), KVM. *see* QEMU (Quick Emulator), KVM

Quick Path Interconnect (QPI), 205–206

R

RAID (redundant array of independent disks)

cache, 417

observability of, 415–416

- overview of, 415
- read-modify-write, 416
- types of, 415–416
- RAM
 - factor analysis, 68
 - as primary memory, 97
- Ramping load, benchmark analysis methodology, 608–611
- Random change anti-method, 37
- Random disk I/O workload, 402–403, 424–425
- Random I/O, 328
- Raw (character) devices, 103
- Raw I/O, 331–332, 418
- read()
 - DTrace to SystemTap examples, 668–670, 672–673
 - function of key system calls, 95
- Reads. *see* RX (receive)
- Read/write (RW) synchronization primitive, 161
- Read/write ratio, disks, 403, 424–425
- Real-time workload, 686
- Reaping
 - defined, 278
 - for freeing memory, 281–282
- Receive. *see* RX (receive)
- Receive buffers, TCP, 492–493, 500
- Receive flow steering (RFS), Linux network stack, 493
- Receive packet steering (RPS), Linux network stack, 493
- Receive side scaling (RSS), Linux network stack, 493
- Redis application, as troubleshooting case study
 - additional information, 634
 - checking for CPU errors, 623
 - checking for disk errors, 625–626
 - checking for memory errors, 624–625
 - checking for network errors, 626
 - comments on process, 633–634
 - DTrace for measuring read latency, 627–629
 - DTrace interrogation of kernel, 629–631
 - getting started in problem solving, 620–622
 - overview of, 617
 - problem statement, 618–619
 - reasons for read latency, 631–633
 - references, 634–635
 - review of performance issues found, 633
 - reviewing available syscalls, 628
 - selecting approach to, 623
- Redundant array of independent disks. *see* RAID (redundant array of independent disks)
- Registers, 686
- Remote disks, 686
- Remote hosts, 477
- Remote memory, 274
- Reno algorithm, for TCP congestion control, 485
- Replay, benchmarking as, 600
- Reporting/archiving current activity. *see* sar (system activity reporter)
- Request for Comments (RFC), 687
- Requests, targets for workload analysis, 34
- Resident memory, 266
- Resident set size (RSS), of allocated main memory, 270, 557
- Resource analysis
 - overview of, 33–34
 - perspectives for performance analysis, 4
- Resource controls
 - allocating disk I/O resources, 429
 - allocating memory, 294
 - comparing virtualization technologies, 582
 - for CPU in OS virtualization, 556–557
 - CPUs and, 222
 - for CPUs in hardware virtualization, 572–573
 - for disk I/O in OS virtualization, 558
 - for file system I/O in OS virtualization, 557–558
 - managing disk or file system I/O, 468
 - for memory capacity in OS virtualization, 557
 - for multitenancy effects, 551
 - network analysis with, 502
 - for network I/O in OS virtualization, 558
 - observability and, 558–563
 - resource management options in OSs, 104
 - strategies for checking, 562–563
 - tuning CPUs, 260
 - tuning memory, 318
 - tuning on Linux, 539
 - tuning on Solaris, 542
- Resource controls facility, 557
- Resource isolation, in cloud computing, 551
- Resources
 - capacity planning and, 66–67
 - cloud computing and limits on, 48
 - examining software resources, 47–48
 - management options in OSs, 104
 - resource list step of USE method, 44–45
 - USE method and, 42
- Response time
 - defined, 16
 - latency and, 18, 35
 - monitoring disk performance, 423–424
 - for storage devices, 399–400

Retransmits, TCP
 monitoring, 498
 tracing, 528–529

Return on investment (ROI), 22, 687

RFC (Request for Comments), 687

RFS (receive flow steering), Linux network stack, 493

Ring buffers, 159, 493

ROI (return on investment), 22, 687

Roles, in systems performance, 2–3

Round-trip time
 defined, 481
 determining route to host, 514–515
 testing network connectivity, 514

Routers
 buffering with, 481
 overview of, 488

Routing, on networks, 476–477

RPS (receive packet steering), Linux network stack, 493

RSS (receive side scaling), Linux network stack, 493

RSS (resident set size), of allocated main memory, 270, 557

Run queues
 defined, 190
 overview of, 192–193
 schedulers and, 98

Run-queue latency, 192

RW (read/write) synchronization primitive, 161

rwsnoop, analyzing file systems, 367

rwtop, analyzing file systems, 367

RX (receive)
 advanced workload characterization/checklist, 497
 defined, 687
 network analysis with USE method, 495
 workload characterization of, 496

S

SACK (selective acknowledgement) algorithm
 for congestion control, 486
 TCP and, 484
 tuning TCP on Linux, 539

Sampling, profiling and, 30

Sanity check, in benchmark analysis, 611

sar (system activity reporter)
 analyzing CPUs, 230
 analyzing file systems, 377–378
 analyzing memory, 298–301
 key options and metrics, 649–650
 monitoring with, 120

 overview of, 509–511
 reporting/archiving disk activity, 440–441
 system-wide counters, 117

SAS (Serial Attached SCSI), 414

SATA (Serial ATA), 414

Saturation
 analyzing CPUs, 196, 216
 analyzing disk devices, 405–406
 analyzing main memory, 271–272
 analyzing networks with USE method, 495
 checking Redis application for memory errors, 624–625
 defined, 16
 indicating with `netstat` on Linux, 504–506
 interpreting metrics of USE method, 48
 measuring network connections, 481
 memory metrics, 293
 metrics, 43, 181–182
 overview of, 29–30
 USE method and, 42

Saturation point, scalability and, 25

Scalability
 Amdahl's Law of Scalability, 60–61
 capacity planning and, 69
 cloud computing, 547–548
 under increasing load, 24–26
 lock analysis and, 182
 modeling in analysis of, 57
 multiple networks providing, 476–477
 multiprocessing/multithreading and, 197–198
 statistical analysis of, 613
 Universal Scalability Law, 61
 visualization of scalability profiles, 59–60

Scalability ceiling, 60

Scaling methods
 applied to CPUs, 223–224
 capacity planning and, 69

Scatter plots
 applied to disks, 462
 for visualizations, 78–79

sched provider, DTrace, 657

Schedulers
 class tuning, 258
 config options, 257–258
 defined, 190
 hypervisor CPU, 572
 key functions of CPU scheduler, 209–210
 latency, 192, 196
 overview of, 98–99
 perf for checking scheduler latency, 245–246
 priority and class schedules, 256–257
 tracing, 242–243

- Scheduling classes
 - kernel support for, 99
 - managing runnable threads, 210–213
 - in real-time, 221
- Scheduling domains, Linux, 214
- Scientific method, 39–41
- Scripts, DTrace
 - advanced network tracking, 531–533
 - overview of, 141–143
- SCSI (Small Computer System Interface)
 - interface, 414
 - tracing SCSI events, 449–450
- Second-level cache, 326
- Sectors
 - defined, 687
 - disk, 396
 - magnetic rotational disks, 409–410
- SEDF (simple earliest deadline first), hypervisor
 - CPU scheduler, 572
- Seek and rotation time, magnetic rotational disks, 408–409
- Segments
 - defined, 266
 - of process address space, 285
- Selective acknowledgement algorithm. *see* SACK (selective acknowledgement) algorithm
- Self-Monitoring Analysis and Reporting Technology (SMART), 460
- Send buffers, TCP, 492–493, 500
- Sequential I/O
 - characterizing disk I/O workload, 424–425
 - disk I/O workload, 402–403
 - overview of, 328
- Serial ATA (SATA), 414
- Serial Attached SCSI (SAS), 414
- Serialization queue (Squeue), GLDv3 software, 491
- Server instances
 - cloud capacity planning with, 548–549
 - cloud computing provisioning framework for, 546
 - defined, 546
 - dynamic sizing in cloud, 549
- Servers, 687
- Service time
 - response time and, 16
 - for storage devices, 399–400
- Shadow page tables, 569–570
- Shards
 - defined, 547–548
 - scaling solution for databases, 69
- Shares, CPU, 556–557
- Shell scripts, 165
- Short-stroking, magnetic rotational disks, 409–410
- Shrinking, freeing memory with, 278
- Simple earliest deadline first (SEDF), hypervisor
 - CPU scheduler, 572
- Simple Network Monitoring Protocol (SNMP), 50, 76
- Simulations
 - as benchmarking type, 599–600
 - inputs for simulation benchmarking, 49–50
 - workload, 57
- Single Root I/O Virtualization (SR-IOV), 570–571
- Slab allocator, 287, 310
- slabtop (slab cache usage)
 - analyzing kernel slab caches related to file systems, 378–379
 - analyzing slab cache usage in Linux, 302
- Sliding window, TCP, 483–484
- Sloth disks, magnetic rotational disks, 411
- Slow-start, TCP, 484
- Slub allocator, 288
- Small Computer System Interface (SCSI)
 - interface, 414
 - tracing SCSI events, 449–450
- SMART (Self-Monitoring Analysis and Reporting Technology), 460
- smartctl, viewing disk health statistics, 460–461
- SmartOS *see also*: Solaris
 - backlog drops, 529–531
 - defined, 687
 - resource controls in OS virtualization, 555
 - retransmit tracing, 529
 - Zones, 48
- SMP (symmetric multiprocessing), 103, 273
- SNMP (Simple Network Monitoring Protocol), 50, 76
- snoop
 - network analysis with, 495
 - overview of, 517–520
 - system-wide tracing, 118
- Sockets
 - analyzing connections, 521–523
 - analyzing duration, 524
 - analyzing internals, 525
 - analyzing I/O, 523–524
 - analyzing latency, 524–525
 - options for tuning network performance, 542
 - tuning buffers on Linux, 537
- sockfs kernel module, Solaris, 491
- soconnect.d script, 522–523

Software

- change management case study, 11–13
- monitoring in OS virtualization, 555
- monitoring resource limits, 66
- tracing, 247–249

Software, CPU-related

- idle threads, 213
- NUMA groups, 214
- overview of, 209
- processor resource-aware, 214
- schedulers, 209–210
- scheduling classes, 210–213

Software, memory-related

- free lists, 280–281
- methods for freeing memory, 278–280
- overview of, 278
- page scanning for freeing memory, 282–284
- reaping for freeing memory, 281–282

Software, network-related

- Linux network stack, 489–490
- network device drivers, 493
- network stack, 488–489
- Solaris network stack, 491
- TCP protocol, 492–493

Software resources

- examining in USE method, 47–48
- USE method checklist in Linux, 641–642
- USE method checklist in Solaris, 646–647

Software stack, diagram of, 1–2

Solaris

- analysis tools, 503
- analyzing file system cache, 377
- analyzing file system statistics, 378
- analyzing kernel memory usage, 302–303, 379
- block device interface and, 420–421
- breakpoint tracing on, 176–177
- comparing Solaris and Linux kernels, 112–113
- configuring network interfaces, 511
- CPU performance tools, 251
- debugging syscall interface, 364
- defined, 687
- dynamic polling, 493
- fault tracing, 311
- free lists, 281
- freeing memory on, 279
- history of Solaris kernel, 106–107
- interface statistics, 507–509
- investigating library calls, 182
- kernel versions and syscall counts, 105
- lgrps (locality groups), 214
- mapping process memory, 307
- measuring VFS latency, 368–369

- memory performance tool, 313
 - memory summary, 380–381
 - microstate accounting, 131
 - monitoring, 120
 - network analysis tools, 503
 - network stack, 491, 493
 - observability sources, 121
 - other network performance tools, 534
 - page scanning methods, 283–284
 - performance features of Solaris kernel, 107–109
 - printing cache activity on UFS, 379–380
 - priority inheritance feature, 197
 - /proc file system, 124–126
 - process binding, 259
 - process status, 304–305
 - processor sets, 222
 - profiling tools, 119
 - resource controls, 104, 260, 318
 - scheduler options, 257–258
 - scheduling classes, 212–213
 - scheduling priority and class, 256–257
 - sources of performance statistics, 127–130
 - summarizing per-disk I/O statistics, 436–440
 - system activity reporter, 300–301, 441, 650
 - TCP fusion on, 483
 - thread-state analysis, 170–171
 - time sharing on, 210
 - top analysis of disk I/O, 453–454
 - top running processes, 562, 624
 - tuning, 539–542
 - tuning memory, 316–317
 - tuning multiple page sizes, 317–318
 - USE method checklist in, 646–647
 - virtual memory statistics, 297–298
 - visibility of block device interface, 443–444
 - zones in OS virtualization, 552
- Solaris IP Datapath Refactoring project, 491
- Solaris Performance and Tools* (McDougall), 50
- Solaris Tunable Parameters Reference Manual*, 539
- Solid-state disks. *see* SSDs (solid-state disks)
- SONET (synchronous optical networking), 687
- Sources, of performance statistics
- Linux delay accounting, 130–131
 - Linux /proc files, 122–124
 - Linux /sys file system, 126–127
 - other sources, 131–133
 - overview of, 120–121
 - /proc file system, 121
 - Solaris kstat files, 127–130
 - Solaris microstate accounting, 131
 - Solaris /proc files, 124–126

- SPARC, 687
- Sparse-root zones model, for overhead in OS virtualization, 555
- SPEC (Standard Performance Evaluation Corporation), 602
- Speedup
 - calculating, 19
 - estimating maximum in approach to latency, 6–7
- Spin locks
 - lock analysis and, 183
 - types of synchronization primitives, 161
- Squeue (serialization queue), GLDv3 software, 491
- SR-IOV (Single Root I/O Virtualization), 570–571
- SSDs (solid-state disks)
 - architecture of, 411–413
 - defined, 687
 - random vs. sequential I/O and, 402
 - types of disks, 395
- Stable providers, DTrace, 520–521
- Stack fishing, 627
- Stacks
 - CPU profiling, 171–173
 - defined, 687
 - how to read, 90
 - overview of, 89–90
 - user and kernel stacks, 90–91
- Stalled execution, of CPU instructions, 194
- Standard deviation, in statistics, 72, 77–78
- Standard Performance Evaluation Corporation (SPEC), 602
- Standards, industry-standard benchmarks, 601–602
- stap
 - analyzing memory, 290
 - analyzing networks, 495
- stat
 - function of key system calls, 95
 - for summarizing CPU cycle behavior, 246–247
- Stateful workload simulation, 600
- Stateless workload simulation, 600
- Static performance tuning
 - checking applications, 185–186
 - checking CPUs, 220–221
 - checking disks, 428–429
 - checking file systems, 359–360
 - checking memory performance, 294
 - checking networks, 501
 - overview of, 55, 501
- Static priority, scheduling classes and, 210
- Static probes, 8
- Static tracing
 - defined, 687
 - with DTrace, 134
- Statistics
 - analyzing benchmark data, 612–613
 - averages, 70–71
 - baselines, 54
 - CoV (coefficient of variation), 72
 - means, 71
 - multimodal distributions and, 73–74
 - outliers, 74
 - overview of, 69
 - quantifying performance issues, 69–70
 - standard deviation, percentiles, and medians, 72
- Storage
 - cloud computing for, 550
 - as performance tuning target, 22
- Storage arrays, 417, 687
- Storage devices
 - disks, 415
 - I/O stack for storage-device based file systems, 101
 - NAS (network-attached storage), 417–418
 - overview of, 415
 - RAID architecture, 415–417
 - response, service, and wait times, 399–400
 - as secondary memory, 97
 - storage arrays, 417, 687
 - storing file system content, 100
- strace
 - analysis phase of drill-down analysis, 51
 - breakpoint tracing on Linux, 173–175
 - debugging syscall interface on Linux, 364–365
 - in event tracing, 54
 - per-process tracing, 118
- Streaming disk I/O workload, 402–403
- Streetlight anti-method, 36–37
- Striping, RAID-0, 415–416
- Stub domains, Xen, 573
- Summary-since-boot values, for monitoring, 76
- Sun Microsystems DTrace tool. *see* DTrace
- Sunny day performance testing, benchmarks for, 599
- Super-serial model, 61
- Surface plots, for visualizations, 80–81
- SUT (system under test), modeling, 17
- Swap area, 266
- Swapping
 - defined, 266
 - Linux methods for freeing memory, 279
 - moving processes between main and secondary memory, 97–98

- Swapping (*continued*)
 - overview of, 271
 - showing swap statistics with `free`, 376
 - Solaris methods for freeing memory, 279
 - thread-state analysis and, 170
 - tools method for checking, 290
 - Switches
 - buffering with, 481
 - overview of, 487–488
 - Symmetric multiprocessing (SMP), 103, 273
 - SYN
 - backlog, 492
 - defined, 687
 - Synchronization primitives, 161
 - Synchronous operations, disk I/O and, 407
 - Synchronous optical networking (SONET), 687
 - Synchronous writes, 331
 - `/sys` file system, Linux, 126–127
 - SysBench tool, for micro-benchmarking, 255, 386
 - Syscall (system calls)
 - analysis, 173
 - breakpoint tracing on Linux, 173–175
 - breakpoint tracing on Solaris, 176–179
 - buffered tracing, 177–179
 - in creating/executing processes, 93
 - debugging syscall interfaces, 364–365
 - defined, 86
 - kernel versions and, 105
 - list of key, 95–96
 - measuring system latency with DTrace, 367–368
 - observability sources, 132
 - overview of, 173
 - performance tuning targets and, 22
 - polling with `poll()` syscall, 159–160
 - syscall (system calls)
 - DTrace to SystemTap examples, 668–672
 - examining Redis application, 627–629
 - syscall provider, DTrace, 651–654
 - `sysctl`, tuning Linux, 536–537
 - sysinfo provider, DTrace, 660–661
 - System activity reporter. *see* `sar` (system activity reporter)
 - System administrators, 33
 - System calls. *see* Syscall (system calls)
 - System design, benchmarking during, 588
 - System logging, 118
 - System performance case study. *see* Redis application, as troubleshooting case study
 - System performance metrics, 27
 - System timer interrupt, 88
 - System under test (SUT), modeling, 17
 - System virtual machine instances, 555
 - Systems performance, introduction
 - activities, 3–4
 - challenging aspect of, 4
 - cloud computing and, 8–9
 - complexity of systems, 5–6
 - dynamic tracing, 7–8
 - latency and, 6–7
 - multiplicity of performance issues, 6
 - overview of, 1–2
 - perspectives, 4
 - roles, 2–3
 - slow disk case study, 9–11
 - software change case study, 11–13
 - subjective nature of performance, 5
 - who's who in, 679–682
 - SystemTap
 - actions and built-ins, 146
 - analyzing memory, 312
 - analyzing networks, 533
 - converting DTrace to. *see* DTrace, converting to SystemTap
 - documentation and resources, 149
 - drill-down analysis, 182
 - dynamic tracing of disks, 451
 - dynamic tracing of file system events, 533
 - dynamic tracing of file systems, 375
 - examples, 146–148
 - overhead, 148
 - overview of, 144–145
 - probes, 145
 - profiling tools, 119
 - system-wide tracing, 118
 - tapsets, 145–146
 - top analysis of disks (`disktop.stp`), 454
 - tracing scheduler events, 243
 - System-wide commands, 561
 - System-wide observability tools
 - counters, 117
 - Linux `/proc` file system, 123
 - overview of, 116
 - profiling, 119
 - Solaris `kstat` files, 127–130
 - tracing, 118
- T**
- Tahoe algorithm, for TCP congestion control, 485
 - Tapsets, grouping SystemTap probes, 145–146
 - Tasklets, interrupt handling and, 92
 - Tasks, 86, 688
 - `taskset`, process binding in Linux, 259

- TCP (Transmission Control Protocol)
 - advanced workload characterization/checklist, 497
 - analyzing networks, 500
 - backlog queues, 492–493
 - buffers, 481, 492–493
 - congestion control, 485
 - connection latency, 19, 480, 484–485
 - event tracing, 525–526
 - improving performance of MTU frames with TCP offload, 479
 - performance features of, 483–484
 - performance monitoring, 498
 - retransmits in Redis application, 620
 - three-way handshakes, 484
 - tuning backlog queue on Linux, 537–538
 - tuning backlog queue on Solaris, 541
 - tuning buffers on Linux, 537
 - tuning congestion control on Linux, 538
 - tuning options on Linux, 538–539
 - tuning options on Solaris, 541
- TCP data transfer time, 19
- TCP fusion, Solaris, 483
- tcp provider, DTrace, 662
- tcp_retransmit_skb() function, DTrace, 528
- tcp_sendmsg(), DTrace, 527–528
- tcp_tw_recycle, tuning TCP on Linux, 539
- tcp_tw_reuse, tuning TCP on Linux, 539
- tcpdump
 - network analysis with, 495
 - overview of, 516–517
 - packet-by-packet inspection with, 53
 - system-wide tracing, 118
 - timestamp used with, 54
- TCP/IP sockets, 483
- TCP/IP stack, 102, 476
- tcpListenDrop, netstat, 508
- tcpListenDropQ0, netstat, 508
- Teams, roles in systems performance, 2
- Technologies, comparison of virtualization, 581–583
- Tenants
 - defined, 546
 - mitigating hardware virtualization overhead, 571–572
 - overhead in OS virtualization, 555
 - resource controls in hardware virtualization, 571
- TENEX, 688
- Thread pools, examining software resources, 47
- Thread-local variables, DTrace, 139
- Thread-state analysis
 - in Linux, 170
 - overview of, 168
 - six states, 168–169
 - in Solaris, 170–171
 - testing effect of software change, 13
 - two states, 168
- Threads/threading
 - defined, 86, 688
 - idle threads, 213
 - interrupt threads, 92
 - load vs. architecture in analysis of performance issues, 24
 - lock analysis applied to multithreaded applications, 182–183
 - multiple threads, 160
 - preemption, 103–104, 196
 - priority inversion, 196–197
 - processes containing one or more threads, 93
 - schedulers and, 98
 - scheduling classes managing runnable threads, 210–213
 - state analysis. *see* Thread-state analysis
- Three-way handshakes, TCP, 484
- Throttle, on resources, 49
- Throughput
 - application performance objectives, 155
 - defined, 16, 688
 - disk I/O workload, 424–425
 - disk terminology, 396
 - magnetic rotational disks, 409
 - metrics of systems performance, 27
 - micro-benchmarking in determination of, 430
 - network analysis with, 494
 - networking terminology, 474
 - performance monitoring of, 498
 - ramping load and measuring, 610
 - read/write ratio and, 403
 - workload characterization of, 496
- Tick latency, 88
- Tick overhead, 88
- Ticks, kernel clock, 88
- time, reporting on CPU usage, 235–236
- Time scales, in analyzing performance, 19–20
- Time series, statistics over time, 74
- Time sharing
 - key functions of CPU scheduler, 209
 - on Linux and Solaris, 210
- Time slices (time quantum), CPU time, 210
- Time to first byte (TTFB) latency, 480
- Time to live (TTL), determining current route to host with traceroute, 514–515
- Time-based metrics, latency as, 19
- Time-based patterns, monitoring, 74–76
- Time-based utilization, 28

Timestamps

- access timestamps, 336
- in event tracing, 54
- iosnoop, 456–457

TIME-WAIT sessions, tuning TCP on Linux, 539

TLB (translation lookaside buffer)

- CPU cache options, 277
- tuning multiple page sizes and, 317

tmp (temporary files), in top-level directories, 100

/tmp file system, 361

Tools method

- applied to CPUs, 215
- applied to disks, 422
- applied to memory, 289–290
- applied to networks, 494–495
- overview of, 41–42

top (top running processes on Linux)

- analyzing CPUs, 231–232
- analyzing file system cache, 376
- analyzing memory, 290
- monitoring, 305
- per-process counters, 117

TPC (Transaction Processing Performance Council), 596–597, 601–602

TPS (transactions per second), 601

Trace file, DTrace to SystemTap example, 672

Trace log replay, 600

traceroute

- applying to Redis application, 618–619
- determining current route to host with traceroute, 514–515
- pathchar vs., 515–516

Tracing

- analyzing file systems, 373–375
- block device events on Linux (blktrace), 457–459
- breakpoint tracing on Linux, 173–175
- breakpoint tracing on Solaris, 176–177
- buffered tracing, 177–179
- disk I/O events, 427–428, 442–444
- disks with iosnoop, 455–457
- event tracing, 53–54
- file systems events, 358–359
- function tracing, 240–241
- memory allocation tracing, 308–311
- memory fault tracing, 311–312
- page-out daemon, 312
- perf for software tracing, 247–249
- per-process observability tools, 131
- scheduler events, 242–243
- SCSI events, 449–450
- slow events, 372–373

static and dynamic, 134–135

tools, 118–119

Tracing, dynamic

- analyzing disks, 450
- analyzing file systems, 373–375
- applying to slow disk case study, 10
- DTrace options, 134–135
- as performance metric, 7–8

Trade-offs, in performance, 20–21

Transaction Processing Performance Council (TPC), 596–597, 601–602

Transactions per second (TPS), 601

Translation lookaside buffer (TLB)

- CPU cache options, 277
- tuning multiple page sizes and, 317

Translation storage buffer (TSB), CPU cache and, 204

Transmission Control Protocol. *see* TCP (Transmission Control Protocol)

Transmit (TX). *see* TX (transmit)

Transmit packet steering (XPS), 493

Transport bus, physical bus used for communication, 396

Traps, 86

truss

- analysis phase of drill-down analysis, 51
- breakpoint tracing on Solaris, 176–177
- debugging syscall interface on Solaris, 364
- in event tracing, 54
- per-process tracing, 118

TSB (translation storage buffer), CPU cache and, 204

TTFB (time to first byte) latency, 480

TTL (time to live), determining current route to host with traceroute, 514–515

Tuning

- benchmarking during, 588
- cache tuning, 55–56, 360
- overview of, 21–22
- static performance tuning. *see* Static performance tuning

Tuning, application-related
static performance tuning, 185–186
targets of, 22

Tuning, CPU-related

- compiler options, 256
- exclusives CPU sets (cpuset), 259
- overview of, 256
- process binding, 259
- processor options (BIOS), 260
- resource controls, 260
- scheduler options, 257–258
- scheduling priority and class, 256–257

- Tuning, disk-related
 - disk controllers, 469–470
 - disk devices, 469
 - operating system, 467–469
 - overview of, 467
 - tunable parameters of operating system, 467–469, 688
 - Tuning, file system-related
 - application calls, 387–388
 - overview of, 387
 - tuning ext systems, 389
 - tuning ZFS systems, 389–391
 - Tuning, memory-related
 - allocators for improving performance of multithreaded applications, 318
 - overview of, 314
 - resource controls, 318
 - tunable parameters, 314–317
 - tunable parameters in memory tuning, 314–317
 - tuning multiple page sizes, 317–318
 - Tuning, network-related
 - Linux, 536–539
 - overview of, 536
 - Solaris, 539–542
 - TX (transmit)
 - advanced workload characterization/checklist, 497
 - defined, 688
 - network analysis with USE method, 495
 - workload characterization of, 496
 - Type 1 hypervisor, 556
 - Type 2 hypervisor, 557
- U**
- UDP (User Datagram Protocol), 486, 497
 - udp provider, DTrace, 663
 - UDS (Unix domain sockets), 483
 - UFS (Unix File System)
 - architecture, 347
 - printing cache activity, 379–380
 - storing file system content, 100
 - UID (user ID), 94
 - UMA (uniform memory access)
 - interconnects and, 204–206
 - main memory architecture, 273
 - Unicast transmission, 477
 - Universal Scalability Law (USL), 61
 - Unix
 - history and Unix kernel, 106
 - memory management, 98
 - overhead in OS virtualization, 554
 - resource controls, 104
 - schedulers, 98
 - Unix domain sockets (UDS), 483
 - Unix File System. *see* UFS (Unix File System)
 - Unknown-unknowns, in performance analysis, 26
 - Unstable providers, DTrace, 520–521
 - uptime analysis, in analyzing CPUs, 224–225
 - USE method
 - analyzing and tuning applications, 181–182
 - analyzing CPUs, 216
 - analyzing disks, 422–423
 - analyzing memory, 290–291
 - analyzing networks, 495–496
 - applying to Redis application, 623–626
 - benchmark analysis with, 607–608
 - checking resource bottlenecks, 10
 - checking software resources, 47–48
 - checking system health, 622
 - cloud computing and, 48
 - creating resource list, 44–45
 - functional block diagrams in, 45
 - Linux checklist for, 637–641
 - metrics, 43–48
 - overview of, 42–43
 - procedure flow in, 44
 - Solaris checklist for, 643–647
 - User Datagram Protocol (UDP), 486, 497
 - User ID (UID), 94
 - User level, 688
 - User mode
 - analysis of time in, 196
 - determining CPU mode of application, 154
 - user programs running in, 89
 - User stacks, 90–91
 - User-level CPU profiling
 - benchmark analysis, 606–607
 - calculating with `prstat`, 234
 - User-space, 86
 - USL (Universal Scalability Law), 61
 - usr (user programs and libraries), in top-level directories, 100
 - Ustack helpers, in DTrace, 172
 - Utilization
 - analyzing applications with USE method, 181–182
 - analyzing CPUs with USE method, 216
 - analyzing networks with USE method, 495
 - application performance objectives, 155
 - averages over time, 71
 - capacity-based, 28–29
 - checking Redis application for memory errors, 624–625
 - of CPUs, 195

Utilization (*continued*)

- defined, 16
- of disk devices, 404–405
- heat maps visualizing, 251–252, 463–465
- of interfaces and, 487
- interpreting metrics of USE method, 48
- of main memory, 271–272
- M/D/1 mean response time vs., 64–65
- memory metrics, 293
- monitoring disk performance, 423–424
- of networks, 482
- non-idle time and, 29
- overview of, 27
- resource analysis focus on, 33
- saturation and, 29–30
- in slow disk case study, 10–11
- in software change case study, 12–13
- surface plots for per-CPU utilization, 80–81
- systems performance metrics, 27
- time-based, 28
- USE method and, 42–43
- of virtual disks, 405

V

- var (varying files), in top-level directories, 100
- Variables, DTrace
 - built-in, 137–138, 667
 - types of, 139–141
- Variance
 - benchmarking, 593
 - CoV (coefficient of variation), 72
- vCPUs (virtual CPUs)
 - CR3 profiling, 579–580
 - resource controls in hardware virtualization, 572
- Vertical perimeters, GLDv3 software, 491
- Vertical scaling, 69
- VFS (virtual file system), 337–338
 - interrogating kernel, 629–631
 - measuring latency with DTrace, 368–370
 - statistics, 363–364
- vfsstat, for virtual file system statistics, 363–364, 562
- Vibration issues, magnetic rotational disks, 410–411
- Virtual disks
 - defined, 396
 - utilization, 405
- Virtual machine control structure (VMCS), 579–580

Virtual machines (VMs), 166

Virtual memory

- available swap, 271–272
- defined, 266
- OSs (operating systems) and, 97
- overview of, 267–268
- states, 270
- vmstat analysis, 226–227

Virtual Memory System (VMS), 688

Virtual memory (VM), 557

Virtual processor, 190

Virtualization

- as basis of cloud computing, 8–9
- hardware virtualization. *see* Hardware virtualization
- OS virtualization. *see* OS virtualization technologies, 581–583

Visual identification, in modeling, 58–60

Visualizations

- of CPUs, 251–254
- of disks, 461
- of distributions, 73–74
- of file systems, 383
- functional block diagrams used in USE method, 45
- heat maps for, 79–80, 462–465
- line charts for, 77–78, 461
- overview of, 76
- scatter plots for, 78–79, 462
- surface plots for, 80–81
- tools for, 81–82

VM (virtual memory), 557

VMCS (virtual machine control structure), 579–580

vminfo provider, DTrace, 661

VMs (virtual machines), 166

VMS (Virtual Memory System), 688

vmstat (virtual memory statistics)

- analyzing CPUs, 226–227
- analyzing file system cache, 376–377
- analyzing memory, 295–298
- checking available memory, 290
- checking Redis application for memory errors, 624
- observability of guest in hardware virtualization, 580
- system-wide counters, 117

VMware ESX, 557

Volumes, file system, 351–352

Voluntary kernel preemption, in Linux, 104

VTune Amplifier XE profiling tool, 119

W

- Wait time, for storage devices, 399–400
- Warm cache, 32
- Web servers, in cloud architecture, 547
- Wireframe models, 80
- Wireshark, for network analysis, 520
- Word size
 - memory performance and, 272
 - processor design and, 198–199
- Work queues, interrupt handling and, 92
- Working set size, micro-benchmarking and, 361
- Workload
 - analysis, 4, 34–35
 - benchmark analysis by ramping, 608–611
 - CPU-bound vs. I/O bound, 99
 - defined, 16
 - latency analysis, 51–52
 - load vs. architecture in analysis of
 - performance issues, 24
 - planning capacity for. *see* Capacity planning
 - scalability under increasing load, 24–26
 - separation, 360
 - simulating, 57
- Workload characterization
 - advanced checklist, 497
 - analyzing and tuning applications, 181
 - applied to CPUs, 216–218
 - applied to disk I/O, 424–426
 - applied to file system, 356–358
 - applied to networks, 496–497
 - benchmark analysis methodology, 608
 - overview of, 49–50
 - studying workload requests, 34
- Workload simulator, testing effect of software change, 11–12

- `write()`, function of key system calls, 95
- Write-back cache, 330, 397
- Writes. *see* TX (transmit)
- Write-through cache, 398

X

- x86, 688
- Xen
 - advanced observability in, 578–579
 - hardware virtualization, 557
 - observability in hardware virtualization for
 - privileged guest/host, 577–578
- xentop tool, 577–578
- xentrace tool, 578
- XPS (transmit packet steering), 493

Z

- ZFS
 - architecture, 348–351
 - exposing internals, 370–371
 - observing pool statistics, 382–383
 - tracing read latency, 371–372
 - tracing slow events, 372–373
 - tuning, 389–391
- ZFS ARC, Solaris methods for freeing memory, 279
- zoneadm property, 557
- Zone-aware commands, 561
- Zone-aware observability tools, 561
- zone.max-physical-memory, 557
- zone.max-swap property, 557
- Zones, Joyent public cloud, 552, 554–555