Memory Systems

Cache, DRAM, Disk

Bruce Jacob
University of Maryland at College Park

Spencer W. Ng
Hitachi Global Storage Technologies

David T. Wang
MetaRAM

With Contributions By

Samuel Rodriguez
Advanced Micro Devices



AMSTERDAM • BOSTON • HEIDELBERG LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann is an imprint of Elsevier



Petitioner Samsung Ex-1009, 0001

Publisher	Denise E.M. Penrose
Acquisitions Editor	Chuck Glaser
Publishing Services Manager	George Morrison
Senior Production Editor	Paul Gottehrer
Developmental Editor	Nate McFadden
Assistant Editor	Kimberlee Honjo
Cover Design	Joanne Blank
Text Design	Dennis Schaefer
Composition	diacriTech
Interior printer	Maple-Vail Book Manufacturing Group
Cover printer	Phoenix Color

Morgan Kaufmann Publishers is an imprint of Elsevier.
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA

This book is printed on acid-free paper.

© 2008 by Elsevier Inc. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, scanning, or otherwise—without prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, E-mail: permissions@elsevier.com. You may also complete your request online via the Elsevier homepage (http://elsevier.com), by selecting "Support & Contact" then "Copyright and Permission" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data
Application submitted

ISBN: 978-0-12-379751-3

For information on all Morgan Kaufmann publications,
visit our Web site at *www.mkp.com* or *www.books.elsevier.com*

Printed in the United States of America
08 09 10 11 12 5 4 3 2 1

Working together to grow
libraries in developing countries
www.elsevier.com | www.bookaid.org | www.sabre.org
ELSEVIER BOOK AID International Sabre Foundation

Petitioner Samsung Ex-1009, 0002

Contents

Preface "It's the Memory, Stupid!" xxxi

Overview On Memory Systems and Their Design 1

Ov.1 Memory Systems 2

Ov.1.1 Locality of Reference Breeds the Memory Hierarchy 2

Ov.1.2 Important Figures of Merit 7

Ov.1.3 The Goal of a Memory Hierarchy 10

Ov.2 Four Anecdotes on Modular Design 14

Ov.2.1 Anecdote I: Systemic Behaviors Exist 15

Ov.2.2 Anecdote II: The DLL in DDR SDRAM 17

Ov.2.3 Anecdote III: A Catch-22 in the Search for Bandwidth 18

Ov.2.4 Anecdote IV: Proposals to Exploit Variability in Cell Leakage 19

Ov.2.5 Perspective 19

Ov.3 Cross-Cutting Issues 20

Ov.3.1 Cost/Performance Analysis 20

Ov.3.2 Power and Energy 26

Ov.3.3 Reliability 32

Ov.3.4 Virtual Memory 34

Ov.4 An Example Holistic Analysis 41

Ov.4.1 Fully-Buffered DIMM vs. the Disk Cache 41

Ov.4.2 Fully Buffered DIMM: Basics 43

Ov.4.3 Disk Caches: Basics 46

Ov.4.4 Experimental Results 47

Ov.4.5 Conclusions 52

Ov.5 What to Expect 54

Petitioner Samsung Ex-1009, 0003

Part I Cache 55

Chapter 1 An Overview of Cache Principles 57

1.1 Caches, 'Caches,' and "Caches" 59

1.2 Locality Principles 62

1.2.1 Temporal Locality 63

1.2.2 Spatial Locality 63

1.2.3 Algorithmic Locality 64

1.2.4 Geographical Locality? Demographical Locality? 65

1.3 What to Cache, Where to Put It, and How to Maintain It 66

1.3.1 Logical Organization Basics: Blocks, Tags, Sets 67

1.3.2 Content Management: To Cache or Not to Cache 68

1.3.3 Consistency Management: Its Responsibilities 69

1.3.4 Inclusion and Exclusion 70

1.4 Insights and Optimizations 73

1.4.1 Perspective 73

1.4.2 Important Issues, Future Directions 77

Chapter 2 Logical Organization 79

2.1 Logical Organization: A Taxonomy 79

2.2 Transparently Addressed Caches 82

2.2.1 Implicit Management: Transparent Caches 86

2.2.2 Explicit Management: Software-Managed Caches 86

2.3 Non-Transparently Addressed Caches 90

2.3.1 Explicit Management: Scratch-Pad Memories 91

2.3.2 Implicit Management: Self-Managed Scratch-Pads 92

2.4 Virtual Addressing and Protection 92

2.4.1 Virtual Caches 93

2.4.2 ASIDs and Protection Bits 96

2.4.3 Inherent Problems 96

2.5 Distributed and Partitioned Caches 97

2.5.1 UMA and NUMA 98

2.5.2 COMA 99

2.5.3 NUCA and NuRAPID 99

2.5.4 Web Caches 100

2.5.5 Buffer Caches 102

Petitioner Samsung Ex-1009, 0004

2.6 Case Studies 103
2.6.1 A Horizontal-Exclusive Organization: Victim Caches, Assist Caches 103
2.6.2 A Software Implementation: BSD's Buffer Cache 104
2.6.3 Another Dynamic Cache Block: Trace Caches 106

Chapter 3 Management of Cache Contents 117
3.1 Case Studies: On-Line Heuristics 120
3.1.1 On-Line Partitioning Heuristics 120
3.1.2 On-Line Prefetching Heuristics 129
3.1.3 On-Line Locality Optimizations 141
3.2 Case Studies: Off-Line Heuristics 151
3.2.1 Off-Line Partitioning Heuristics 151
3.2.2 Off-Line Prefetching Heuristics 155
3.2.3 Off-Line Locality Optimizations 161
3.3 Case Studies: Combined Approaches 169
3.3.1 Combined Approaches to Partitioning 170
3.3.2 Combined Approaches to Prefetching 174
3.3.3 Combined Approaches to Optimizing Locality 180
3.4 Discussions 202
3.4.1 Proposed Scheme vs. Baseline 202
3.4.2 Prefetching vs. Locality Optimizations 203
3.4.3 Application-Directed Management vs. Transparent Management 203
3.4.4 Real Time vs. Average Case 204
3.4.5 Naming vs. Cache Conflicts 205
3.4.6 Dynamic vs. Static Management 208
3.5 Building a Content-Management Solution 212
3.5.1 Degree of Dynamism 212
3.5.2 Degree of Prediction 213
3.5.3 Method of Classification 213
3.5.4 Method for Ensuring Availability 214

Chapter 4 Management of Cache Consistency 217
4.1 Consistency with Backing Store 218
4.1.1 Write-Through 218
4.1.2 Delayed Write, Driven By the Cache 219
4.1.3 Delayed Write, Driven by Backing Store 220

Petitioner Samsung Ex-1009, 0005

4.2 Consistency with Self 220
4.2.1 Virtual Cache Management 220
4.2.2 ASID Management 225
4.3 Consistency with Other Clients 226
4.3.1 Motivation, Explanation, Intuition 226
4.3.2 Coherence vs. Consistency 231
4.3.3 Memory-Consistency Models 233
4.3.4 Hardware Cache-Coherence Mechanisms 240
4.3.5 Software Cache-Coherence Mechanisms 254

Chapter 5 Implementation Issues 257
5.1 Overview 257
5.2 SRAM Implementation 258
5.2.1 Basic 1-Bit Memory Cell 259
5.2.2 Address Decoding 262
5.2.3 Physical Decoder Implementation 268
5.2.4 Peripheral Bitline Circuits 278
5.2.5 Sense Amplifiers 281
5.2.6 Write Amplifier 283
5.2.7 SRAM Partitioning 285
5.2.8 SRAM Control and Timing 286
5.2.9 SRAM Interface 292
5.3 Advanced SRAM Topics 293
5.3.1 Low-Leakage Operation 293
5.4 Cache Implementation 297
5.4.1 Simple Caches 297
5.4.2 Processor Interfacing 298
5.4.3 Multiporting 298

Chapter 6 Cache Case Studies 301
6.1 Logical Organization 301
6.1.1 Motorola MPC7450 301
6.1.2 AMD Opteron 301
6.1.3 Intel Itanium-2 303

Petitioner Samsung Ex-1009, 0006

6.2 Pipeline Interface 304
6.2.1 Motorola MPC7450 304
6.2.2 AMD Opteron 304
6.2.3 Intel Itanium-2 304
6.3 Case Studies of Detailed Itanium-2 Circuits 305
6.3.1 L1 Cache RAM Cell Array 305
6.3.2 L2 Array Bitline Structure 305
6.3.3 L3 Subarray Implementation 308
6.3.4 Itanium-2 TLB and CAM Implementation 308

Part II DRAM 313

Chapter 7 Overview of DRAMs 315

7.1 DRAM Basics: Internals, Operation 316
7.2 Evolution of the DRAM Architecture 322
7.2.1 Structural Modifications Targeting Throughput 322
7.2.2 Interface Modifications Targeting Throughput 328
7.2.3 Structural Modifications Targeting Latency 330
7.2.4 Rough Comparison of Recent DRAMs 331
7.3 Modern-Day DRAM Standards 332
7.3.1 Salient Features of JEDEC's SDRAM Technology 332
7.3.2 Other Technologies, Rambus in Particular 335
7.3.3 Comparison of Technologies in Rambus and JEDEC DRAM 341
7.3.4 Alternative Technologies 343
7.4 Fully Buffered DIMM: A Compromise of Sorts 348
7.5 Issues in DRAM Systems, Briefly 350
7.5.1 Architecture and Scaling 350
7.5.2 Topology and Timing 350
7.5.3 Pin and Protocol Efficiency 351
7.5.4 Power and Heat Dissipation 351
7.5.5 Future Directions 351

Petitioner Samsung Ex-1009, 0007

Chapter 8 DRAM Device Organization: Basic Circuits and Architecture 353
8.1 DRAM Device Organization 353
8.2 DRAM Storage Cells 355
8.2.1 Cell Capacitance, Leakage, and Refresh 356
8.2.2 Conflicting Requirements Drive Cell Structure 356
8.2.3 Trench Capacitor Structure 357
8.2.4 Stacked Capacitor Structure 357
8.3 RAM Array Structures 358
8.3.1 Open Bitline Array Structure 359
8.3.2 Folded Bitline Array Structure 360
8.4 Differential Sense Amplifier 360
8.4.1 Functionality of Sense Amplifiers in DRAM Devices 361
8.4.2 Circuit Diagram of a Basic Sense Amplifier 362
8.4.3 Basic Sense Amplifier Operation 362
8.4.4 Voltage Waveform of Basic Sense Amplifier Operation 363
8.4.5 Writing into DRAM Array 365
8.5 Decoders and Redundancy 366
8.5.1 Row Decoder Replacement Example 368
8.6 DRAM Device Control Logic 368
8.6.1 Synchronous vs. Non-Synchronous 369
8.6.2 Mode Register-Based Programmability 370
8.7 DRAM Device Configuration 370
8.7.1 Device Configuration Trade-offs 371
8.8 Data I/O 372
8.8.1 Burst Lengths and Burst Ordering 372
8.8.2 N-Bit Prefetch 372
8.9 DRAM Device Packaging 373
8.10 DRAM Process Technology and Process Scaling Considerations 374
8.10.1 Cost Considerations 375
8.10.2 DRAM- vs. Logic-Optimized Process Technology 375

Chapter 9 DRAM System Signaling and Timing 377
9.1 Signaling System 377

Petitioner Samsung Ex-1009, 0008

9.2 Transmission Lines on PCBs 379
9.2.1 Brief Tutorial on the Telegrapher's Equations 380
9.2.2 RC and LC Transmission Line Models 382
9.2.3 LC Transmission Line Model for PCB Traces 383
9.2.4 Signal Velocity on the LC Transmission Line 383
9.2.5 Skin Effect of Conductors 384
9.2.6 Dielectric Loss 384
9.2.7 Electromagnetic Interference and Crosstalk 386
9.2.8 Near-End and Far-End Crosstalk 387
9.2.9 Transmission Line Discontinuities 388
9.2.10 Multi-Drop Bus 390
9.2.11 Socket Interfaces 391
9.2.12 Skew 392
9.2.13 Jitter 392
9.2.14 Inter-Symbol Interference (ISI) 393
9.3 Termination 393
9.3.1 Series Stub (Serial) Termination 394
9.3.2 On-Die (Parallel) Termination 394
9.4 Signaling 395
9.4.1 Eye Diagrams 396
9.4.2 Low-Voltage TTL (Transistor-Transistor Logic) 396
9.4.3 Voltage References 398
9.4.4 Series Stub Termination Logic 398
9.4.5 RSL and DRSL 399
9.5 Timing Synchronization 399
9.5.1 Clock Forwarding 400
9.5.2 Phase-Locked Loop (PLL) 401
9.5.3 Delay-Locked Loop (DLL) 402
9.6 Selected DRAM Signaling and Timing Issues 402
9.6.1 Data Read and Write Timing in DDRx SDRAM Memory Systems 404
9.6.2 The Use of DLL in DDRx SDRAM Devices 406
9.6.3 The Use of PLL in XDR DRAM Devices 406
9.7 Summary 408

Petitioner Samsung Ex-1009, 0009

Chapter 10 DRAM Memory System Organization.....409
10.1 Conventional Memory System 409
10.2 Basic Nomenclature 409
10.2.1 Channel 410
10.2.2 Rank 413
10.2.3 Bank 414
10.2.4 Row 415
10.2.5 Column 415
10.2.6 Memory System Organization: An Example 416
10.3 Memory Modules 417
10.3.1 Single In-line Memory Module (SIMM) 418
10.3.2 Dual In-line Memory Module (DIMM) 418
10.3.3 Registered Memory Module (RDIMM) 418
10.3.4 Small Outline DIMM (SO-DIMM) 419
10.3.5 Memory Module Organization 420
10.3.6 Serial Presence Detect (SPD) 421
10.4 Memory System Topology 422
10.4.1 Direct RDRAM System Topology 422
10.5 Summary 423

Chapter 11 Basic DRAM Memory-Access Protocol 425
11.1 Basic DRAM Commands 425
11.1.1 Generic DRAM Command Format 427
11.1.2 Summary of Timing Parameters 427
11.1.3 Row Access Command 428
11.1.4 Column-Read Command 429
11.1.5 Column-Write Command 430
11.1.6 Precharge Command 431
11.1.7 Refresh Command 431
11.1.8 A Read Cycle 433
11.1.9 A Write Cycle 434
11.1.10 Compound Commands 434
11.2 DRAM Command Interactions 436
11.2.1 Consecutive Reads and Writes to Same Rank 437
11.2.2 Read to Precharge Timing 438
11.2.3 Consecutive Reads to Different Rows of Same Bank 438

Petitioner Samsung Ex-1009, 0010

11.2.4 Consecutive Reads to Different Banks: Bank Conflict *440*
11.2.5 Consecutive Read Requests to Different Ranks *441*
11.2.6 Consecutive Write Requests: Open Banks *442*
11.2.7 Consecutive Write Requests: Bank Conflicts *444*
11.2.8 Write Request Following Read Request: Open Banks *444*
11.2.9 Write Request Following Read Request to Different Banks, Bank Conflict, Best Case, No Reordering *445*
11.2.10 Read Following Write to Same Rank, Open Banks *446*
11.2.11 Write to Precharge Timing *447*
11.2.12 Read Following Write to Different Ranks, Open Banks *447*
11.2.13 Read Following Write to Same Bank, Bank Conflict *448*
11.2.14 Read Following Write to Different Banks of Same Rank, Bank Conflict, Best Case, No Reordering *449*
11.2.15 Column-Read-and-Precharge Command Timing *449*
11.2.16 Column-Write-and-Precharge Timing *450*
11.3 Additional Constraints 450
11.3.1 Device Power Limit *450*
11.3.2 t_{RRD}: Row-to-Row (Activation) Delay *452*
11.3.3 t_{FAW}: Four-Bank Activation Window *453*
11.3.4 2T Command Timing in Unbuffered Memory Systems *454*
11.4 Command Timing Summary 454
11.5 Summary 454

Chapter 12 Evolutionary Developments of DRAM Device Architecture 457
12.1 DRAM Device Families 457
12.1.1 Cost (Capacity), Latency, Bandwidth, and Power *457*
12.2 Historical-Commodity DRAM Devices 458
12.2.1 The Intel 1103 ..... *459*
12.2.2 Asynchronous DRAM Devices *461*
12.2.3 Page Mode and Fast Page Mode DRAM (FPM DRAM) *461*
12.2.4 Extended Data-Out (EDO) and Burst Extended Data-Out (BEDO) Devices *463*
12.3 Modern-Commodity DRAM Devices 464
12.3.1 Synchronous DRAM (SDRAM) *465*

Petitioner Samsung Ex-1009, 0011

12.3.2 Double Data Rate SDRAM (DDR SDRAM) *471*
12.3.3 DDR2 SDRAM *474*
12.3.4 Protocol and Architectural Differences *475*
12.3.5 DDR3 SDRAM *476*
12.3.6 Scaling Trends of Modern-Commodity DRAM Devices *477*
12.4 High Bandwidth Path 480
12.4.1 Direct RDRAM *480*
12.4.2 Technical and Pseudo-Technical Issues of Direct RDRAM *487*
12.4.3 XDR Memory System *491*
12.5 Low Latency 494
12.5.1 Reduced Latency DRAM (RLDRAM) *494*
12.5.2 Fast Cycle DRAM (FCRAM) *495*
12.6 Interesting Alternatives 495
12.6.1 Virtual Channel Memory (VCDRAM) *495*
12.6.2 Enhanced SDRAM (ESDRAM) *496*

Chapter 13 DRAM Memory Controller 497

13.1 DRAM Controller Architecture 497
13.2 Row-Buffer-Management Policy 499
13.2.1 Open-Page Row-Buffer-Management Policy *499*
13.2.2 Close-Page Row-Buffer-Management Policy *499*
13.2.3 Hybrid (Dynamic) Row-Buffer-Management Policies *500*
13.2.4 Performance Impact of Row-Buffer-Management Policies *500*
13.2.5 Power Impact of Row-Buffer-Management Policies *501*
13.3 Address Mapping (Translation) 502
13.3.1 Available Parallelism in Memory System Organization *503*
13.3.2 Parameter of Address Mapping Schemes *504*
13.3.3 Baseline Address Mapping Schemes *505*
13.3.4 Parallelism vs. Expansion Capability *506*
13.3.5 Address Mapping in the Intel 82955X MCH *506*
13.3.6 Bank Address Aliasing (Stride Collision) *510*
13.4 Performance Optimization 511
13.4.1 Write Caching *512*
13.4.2 Request Queue Organizations *513*
13.4.3 Refresh Management *514*

Petitioner Samsung Ex-1009, 0012

13.4.4 Agent-Centric Request Queuing Organization *516*
13.4.5 Feedback-Directed Scheduling *518*
13.5 Summary **518**

Chapter 14 The Fully Buffered DIMM Memory System **519**
14.1 Introduction **519**
14.2 Architecture **521**
14.3 Signaling and Timing **524**
14.3.1 Clock Data Recovery *524*
14.3.2 Unit Interval *525*
14.3.3 Resample and Resync *525*
14.4 Access Protocol **526**
14.4.1 Frame Definitions *527*
14.4.2 Command Definitions *528*
14.4.3 Frame and Command Scheduling *528*
14.5 The Advanced Memory Buffer **530**
14.5.1 SMBus Interface *531*
14.5.2 Built-In Self-Test (BIST) *532*
14.5.3 Thermal Sensor *532*
14.6 Reliability, Availability, and Serviceability **532**
14.6.1 Checksum Protection in the Transport Layer *532*
14.6.2 Bit Lane Steering *533*
14.6.3 Fail-over Modes *534*
14.6.4 Hot Add and Replace *534*
14.7 FBDIMM Performance Characteristics **535**
14.7.1 Fixed vs. Variable Latency Scheduling *538*
14.8 Perspective **540**

Chapter 15 Memory System Design Analysis **541**
15.1 Overview **541**
15.2 Workload Characteristics **543**
15.2.1 164.gzip: C Compression *544*
15.2.2 176.gcc: C Programming Language Compiler *545*
15.2.3 197.parser: C Word Processing *545*
15.2.4 255.vortex: C Object-Oriented Database *546*

Petitioner Samsung Ex-1009, 0013

15.2.5 172.mgrid: Fortran 77 Multi-Grid Solver: 3D Potential Field 547
15.2.6 SETI@HOME 547
15.2.7 Quake 3 548
15.2.8 178.galgel, 179.art, 183.equake, 188.ammp, JMark 2.0, and 3DWinbench 548
15.2.9 Summary of Workload Characteristics 550
15.3 The RAD Analytical Framework 551
15.3.1 DRAM-Access Protocol 551
15.3.2 Computing DRAM Protocol Overheads 551
15.3.3 Computing Row Cycle Time Constraints 553
15.3.4 Computing Row-to-Row Activation Constraints 555
15.3.5 Request Access Distance Efficiency Computation 557
15.3.6 An Applied Example for a Close-Page System 558
15.3.7 An Applied Example for an Open-Page System 558
15.3.8 System Configuration for RAD-Based Analysis 559
15.3.9 Open-Page Systems: 164.gzip 561
15.3.10 Open-Page Systems: 255.vortex 562
15.3.11 Open-Page Systems: Average of All Workloads 562
15.3.12 Close-Page Systems: 164.gzip 565
15.3.13 Close-Page Systems: SETI@HOME Processor Bus Trace 566
15.3.14 Close-Page Systems: Average of All Workloads 567
15.3.15 t_{FAW} Limitations in Open-Page System: All Workloads 568
15.3.16 Bandwidth Improvements: 8-Banks vs. 16-Banks 568
15.4 Simulation-Based Analysis 570
15.4.1 System Configurations 570
15.4.2 Memory Controller Structure 571
15.4.3 DRAM Command Scheduling Algorithms 572
15.4.4 Workload Characteristics 575
15.4.5 Timing Parameters 575
15.4.6 Protocol Table 575
15.4.7 Queue Depth, Scheduling Algorithms, and Burst Length 577
15.4.8 Effect of Burst Length on Sustainable Bandwidth 578
15.4.9 Burst Chop in DDR3 SDRAM Devices 579
15.4.10 Revisiting the 8-Bank and 16-Bank Issue with DRAMSim 584
15.4.11 8 Bank vs. 16 Banks — Relaxed t_{FAW} and t_{WTR} 587
15.4.12 Effect of Transaction Ordering on Latency Distribution 587

Petitioner Samsung Ex-1009, 0014

15.5 A Latency-Oriented Study 590
15.5.1 Experimental Framework 590
15.5.2 Simulation Input 592
15.5.3 Limit Study: Latency Bandwidth Characteristics 592
15.5.4 Latency 593
15.6 Concluding Remarks 596

Part III Disk 599

Chapter 16 Overview of Disks 601

16.1 History of Disk Drives 601
16.1.1 Evolution of Drives 603
16.1.2 Areal Density Growth Trend 605
16.2 Principles of Hard Disk Drives 606
16.2.1 Principles of Rotating Storage Devices 607
16.2.2 Magnetic Rotating Storage Device—Hard Disk Drive 608
16.3 Classifications of Disk Drives 609
16.3.1 Form Factor 609
16.3.2 Application 609
16.3.3 Interface 609
16.4 Disk Performance Overview 610
16.4.1 Disk Performance Metrics 610
16.4.2 Workload Factors Affecting Performance 612
16.4.3 Video Application Performance 612
16.5 Future Directions in Disks 612

Chapter 17 The Physical Layer 615

17.1 Magnetic Recording 615
17.1.1 Ferromagnetism 615
17.1.2 Magnetic Fields 616
17.1.3 Hysteresis Loop 618
17.1.4 Writing 618
17.1.5 Reading 620

Petitioner Samsung Ex-1009, 0015

17.2 Mechanical and Magnetic Components 620
17.2.1 Disks 621
17.2.2 Spindle Motor 623
17.2.3 Heads 625
17.2.4 Slider and Head-Gimbal Assembly 631
17.2.5 Head-Stack Assembly and Actuator 633
17.2.6 Multiple Platters 635
17.2.7 Start/Stop 636
17.2.8 Magnetic Disk Recording Integration 637
17.2.9 Head-Disk Assembly 639
17.3 Electronics 640
17.3.1 Controller 640
17.3.2 Memory 642
17.3.3 Recording Channel 642
17.3.4 Motor Controls 646

Chapter 18 The Data Layer 649
18.1 Disk Blocks and Sectors 649
18.1.1 Fixed-Size Blocks 649
18.1.2 Variable Size Blocks 650
18.1.3 Sectors 650
18.2 Tracks and Cylinders 652
18.3 Address Mapping 654
18.3.1 Internal Addressing 654
18.3.2 External Addressing 654
18.3.3 Logical Address to Physical Location Mapping 655
18.4 Zoned-Bit Recording 658
18.4.1 Handling ZBR 661
18.5 Servo 662
18.5.1 Dedicated Servo 662
18.5.2 Embedded Servo 663
18.5.3 Servo ID and Seek 666
18.5.4 Servo Burst and Track Following 667
18.5.5 Anatomy of a Servo 669
18.5.6 ZBR and Embedded Servo 669

Petitioner Samsung Ex-1009, 0016

18.6 Sector ID and No-ID Formatting 670
18.7 Capacity 672
18.8 Data Rate 673
18.9 Defect Management 673
18.9.1 Relocation Schemes 674
18.9.2 Types of Defects 675
18.9.3 Error Recovery Procedure 676

Chapter 19 Performance Issues and Design Trade-Offs 677

19.1 Anatomy of an I/O 677
19.1.1 Adding It All Up 679
19.2 Some Basic Principles 681
19.2.1 Effect of User Track Capacity 681
19.2.2 Effect of Cylinder Capacity 682
19.2.3 Effect of Track Density 684
19.2.4 Effect of Number of Heads 686
19.3 BPI vs. TPI 688
19.4 Effect of Drive Capacity 689
19.4.1 Space Usage Efficiency 690
19.4.2 Performance Implication 690
19.5 Concentric Tracks vs. Spiral Track 692
19.5.1 Optical Disks 693
19.6 Average Seek 694
19.6.1 Disks Without ZBR 694
19.6.2 Disks With ZBR 695

Chapter 20 Drive Interface 699

20.1 Overview of Interfaces 699
20.1.1 Components of an Interface 701
20.1.2 Desirable Characteristics of Interface 701
20.2 ATA 702
20.3 Serial ATA 703
20.4 SCSI 705
20.5 Serial SCSI 706
20.6 Fibre Channel 707
20.7 Cost, Performance, and Reliability 709

Petitioner Samsung Ex-1009, 0017

Chapter 21 Operational Performance Improvement 711
21.1 Latency Reduction Techniques 711
21.1.1 Dual Actuator 711
21.1.2 Multiple Copies 712
21.1.3 Zero Latency Access 713
21.2 Command Queueing and Scheduling 715
21.2.1 Seek Time-Based Scheduling 716
21.2.2 Total Access Time Based Scheduling 717
21.2.3 Sequential Access Scheduling 723
21.3 Reorganizing Data on the Disk 723
21.3.1 Defragmentation 724
21.3.2 Frequently Accessed Files 724
21.3.3 Co-Locating Access Clusters 725
21.3.4 ALIS 726
21.4 Handling Writes 728
21.4.1 Log-Structured Write 728
21.4.2 Disk Buffering of Writes 729
21.5 Data Compression 729

Chapter 22 The Cache Layer 731
22.1 Disk Cache 731
22.1.1 Why Disk Cache Works 731
22.1.2 Cache Automation 732
22.1.3 Read Cache, Write Cache 732
22.2 Cache Organizations 735
22.2.1 Desirable Features of Cache Organization 735
22.2.2 Fixed Segmentation 736
22.2.3 Circular Buffer 737
22.2.4 Virtual Memory Organization 738
22.3 Caching Algorithms 741
22.3.1 Perspective of Prefetch 741
22.3.2 Lookahead Prefetch 742
22.3.3 Look-behind Prefetch 742
22.3.4 Zero Latency Prefetch 743

Petitioner Samsung Ex-1009, 0018

22.3.5 ERP During Prefetch ... *743*
22.3.6 Handling of Sequential Access ... *743*
22.3.7 Replacement Policies ... *745*

Chapter 23 Performance Testing ... 747
23.1 Test and Measurement ... 747
23.1.1 Test Initiator ... *747*
23.1.2 Monitoring and Measuring ... *749*
23.1.3 The Test Drive ... *750*
23.2 Basic Tests ... 750
23.2.1 Media Data Rate ... *751*
23.2.2 Disk Buffer Data Rate ... *751*
23.2.3 Sequential Performance ... *752*
23.2.4 Random Performance ... *752*
23.2.5 Command Reordering Performance ... *755*
23.3 Benchmark Tests ... 755
23.3.1 Guidelines for Benchmarking ... *756*
23.4 Drive Parameters Tests ... 757
23.4.1 Geometry and More ... *757*
23.4.2 Seek Time ... *759*

Chapter 24 Storage Subsystems ... 763
24.1 Data Striping ... 763
24.2 Data Mirroring ... 765
24.2.1 Basic Mirroring ... *766*
24.2.2 Chained Decluster Mirroring ... *766*
24.2.3 Interleaved Decluster Mirroring ... *767*
24.2.4 Mirroring Performance Comparison ... *767*
24.2.5 Mirroring Reliability Comparison ... *769*
24.3 RAID ... 770
24.3.1 RAID Levels ... *770*
24.3.2 RAID Performance ... *774*
24.3.3 RAID Reliability ... *775*
24.3.4 Sparing ... *776*

Petitioner Samsung Ex-1009, 0019

24.3.5 RAID Controller *778*
24.3.6 Advanced RAIDs *779*
24.4 SAN **780**
24.5 NAS **782**
24.6 ISCSI **783**

Chapter 25 Advanced Topics 785

25.1 Perpendicular Recording **785**
25.1.1 Write Process, Write Head, and Media *786*
25.1.2 Read Process and Read Head *788*
25.2 Patterned Media **788**
25.2.1 Fully Patterned Media *789*
25.2.2 Discrete Track Media *790*
25.3 Thermally Assisted Recording **790**
25.4 Dual Stage Actuator **792**
25.4.1 Microactuators *793*
25.5 Adaptive Formatting **794**
25.6 Hybrid Disk Drive **796**
25.6.1 Benefits *796*
25.6.2 Architecture *796*
25.6.3 Proposed Interface *797*
25.7 Object-Based Storage **798**
25.7.1 Object Storage Main Concept *798*
25.7.2 Object Storage Benefits *800*

Chapter 26 Case Study 803

26.1 The Mechanical Components **803**
26.1.1 Seek Profile *804*
26.2 Electronics **806**
26.3 Data Layout **806**
26.3.1 Data Rate *808*
26.4 Interface **809**
26.5 Cache **810**
26.6 Performance Testing **810**

Petitioner Samsung Ex-1009, 0020

26.6.1 Sequential Access ... *810*
26.6.2 Random Access ... *811*

Part IV Cross-Cutting Issues ... 813

Chapter 27 The Case for Holistic Design ... 815

27.1 Anecdotes, Revisited ... **816**
27.1.1 Anecdote I: Systemic Behaviors Exist ... *816*
27.1.2 Anecdote II: The DLL in DDR SDRAM ... *818*
27.1.3 Anecdote III: A Catch-22 in the Search for Bandwidth ... *822*
27.1.4 Anecdote IV: Proposals to Exploit Variability in Cell Leakage ... *824*
27.2 Perspective ... **827**

Chapter 28 Analysis of Cost and Performance ... 829

28.1 Combining Cost and Performance ... **829**
28.2 Pareto Optimality ... **830**
28.2.1 The Pareto-Optimal Set: An Equivalence Class ... *830*
28.2.2 Stanley's Observation ... *832*
28.3 Taking Sampled Averages Correctly ... **833**
28.3.1 Sampling Over Time ... *834*
28.3.2 Sampling Over Distance ... *835*
28.3.3 Sampling Over Fuel Consumption ... *836*
28.3.4 The Moral of the Story ... *837*
28.4 Metrics for Computer Performance ... **838**
28.4.1 Performance and the Use of Means ... *838*
28.4.2 Problems with Normalization ... *839*
28.4.3 The Meaning of Performance ... *842*
28.5 Analytical Modeling and the Miss-Rate Function ... **843**
28.5.1 Analytical Modeling ... *843*
28.5.2 The Miss-Rate Function ... *844*

Chapter 29 Power and Leakage ... 847

29.1 Sources of Leakage in CMOS Devices ... **847**
29.2 A Closer Look at Subthreshold Leakage ... **851**

Petitioner Samsung Ex-1009, 0021

29.3 υCACTI and Energy/Power Breakdown of Pipelined Nanometer Caches 855
29.3.1 Leakage in SRAM Cells 855
29.3.2 Pipelined Caches 857
29.3.3 Modeling 858
29.3.4 Dynamic and Static Power 859
29.3.5 Detailed Power Breakdown 860

Chapter 30 Memory Errors and Error Correction 865

30.1 Types and Causes of Failures 865
30.1.1 Alpha Particles 866
30.1.2 Primary Cosmic Rays and Terrestrial Neutrons 866
30.1.3 Soft Error Mechanism 867
30.1.4 Single-Bit and Multi-Bit Failures 867
30.2 Soft Error Rates and Trends 868
30.3 Error Detection and Correction 869
30.3.1 Parity 869
30.3.2 Single-Bit Error Correction (SEC ECC) 870
30.3.3 Single-Bit Error Correction, Double-Bit Error Detection (SECDED ECC) 873
30.3.4 Multi-Bit Error Detection and Correction: Bossen's b-Adjacent Algorithm 874
30.3.5 Bit Steering and Chipkill 875
30.3.6 Chipkill with ×8 DRAM Devices 877
30.3.7 Memory Scrubbing 879
30.3.8 Bullet Proofing the Memory System 880
30.4 Reliability of Non-DRAM Systems 880
30.4.1 SRAM 880
30.4.2 Flash 880
30.4.3 MRAM 880
30.5 Space Shuttle Memory System 881

Chapter 31 Virtual Memory 883

31.1 A Virtual Memory Primer 884
31.1.1 Address Spaces and the Main Memory Cache 885
31.1.2 Address Mapping and the Page Table 886
31.1.3 Hierarchical Page Tables 887

Petitioner Samsung Ex-1009, 0022

31.1.4 Inverted Page Tables *890*
31.1.5 Comparison: Inverted vs. Hierarchical *892*
31.1.6 Translation Lookaside Buffers, Revisited *893*
31.1.7 Perspective: Segmented Addressing Solves the Synonym Problem *895*
31.1.8 Perspective: A Taxonomy of Address Space Organizations *901*
31.2 Implementing Virtual Memory 906
31.2.1 The Basic In-Order Pipe *908*
31.2.2 Precise Interrupts in Pipelined Computers *910*

References 921

Index 955

Petitioner Samsung Ex-1009, 0023

CHAPTER 1

An Overview of Cache Principles

A cache (also called "look-aside" [Bloom et al. 1962] or "slave memory" [Wilkes 1965]) can speed up accesses to all manners of storage devices, including tape drives, disk drives, main memory, servers on the network (e.g., web servers are a type of storage device), and even other caches. It works on the principle of *locality of reference*, the tendency of applications to reference a predictably small amount of data within a given window of time [Belady 1966, Denning 1970]. Any storage device can be characterized by its access time and cost per bit, where faster storage technologies tend to have a lower access time and cost more per bit than slower technologies. A cache fronting a given storage device would be built from a technology that is faster than that of the storage device in question; the cache technology would typically cost more on a per-bit basis, but to be effective the cache would only need to be large enough to hold the application's *working set*—the set of instructions and/or data items the application is currently using to perform its computations [Denning 1967]. Due to locality of reference, most of the application accesses will be satisfied out of the cache, and so most of the time the access characteristics will be that of the cache: far faster and often consuming less energy than the larger storage device behind the cache.

As far as implementation technologies, caches can be built of anything that holds state. Hardware caches are used widely in computer systems (SRAM can act as a cache for DRAM; DRAM can act as a cache for disk; fast DRAM can act as a cache for slow DRAM; disk can act as a cache for tape; etc.), and software caches abound as well (the operating system caches disk blocks and network addresses; web proxies cache popular web-based documents; even hardware devices such as disk drives cache data locally, implementing the cache algorithm in software executing on a control processor). In addition, what is logically a single cache may, in fact, be comprised of many distinct entities that operate in concert. For instance, the separate last-level caches belonging to the different processors in a symmetric multiprocessor can be thought of (and even managed) as a distributed but logically monolithic entity. Conversely, some cache organizations described as a single cache are, in fact, multi-level hierarchies [Kim et al. 2002]. Neither of these scenarios are erroneous or should be taken as misleading. Any set of multiple storage units can be labeled and analyzed as a single "cache," and any monolithic cache can be labeled and analyzed as a set of multiple, independent entities. The choice of how to portray a particular cache is often made out of necessity (e.g., to make an algorithm feasible, as in Wheeler and Bershad's treatment of a single cache as a collection of logically independent partitions [1992]) or out of convenience (e.g., to simplify an explanation, as in Kim's NUCA [Non-Uniform Cache Architecture]). The bottom line is that the term *cache* means many things to many different fields of discipline, and rather than treat some uses as "correct" and others as "incorrect," we consider it quite useful to develop a framework that encompasses most, if not all, uses of the term. The primary contribution of Part I of the book is the presentation of just such a framework.

Caches can be *transparent*, meaning that they work independently of the client making the request (e.g., the application software executing on the computer) and therefore have built-in heuristics that determine what data to retain. Caches can also be

Petitioner Samsung Ex-1009, 0024

managed explicitly by the client making the request (these are often called *scratch-pads*) or can represent some hybrid of the two extremes. Examples of transparent caches include the processor caches found in general-purpose systems, the file caches found in most distributed file systems, and nearly all forms of web cache given that the "client" of a web cache is typically the person using the web browser. Examples of scratch-pads include the ubiquitous register file as well as the tagless SRAMs found in nearly all microcontrollers and digital signal processors. The former (transparent cache) operates autonomously with respect to the client; the cached version can be deleted, lost, or invalidated, and the original is still available in the backing store. The latter (scratch-pad) requires the direct control of the client to move data in and out; working with the *cached* version of a datum means one is, in fact, working with the *only* version—the cache is not autonomous and therefore makes no attempt to propagate changes back to the original.

Hybrid caches incorporate aspects of both autonomy and client-directed management. Examples include Cheriton's software-controlled caches in the VMP multiprocessor system [Cheriton et al. 1986, 1988, 1989], as well as modern "snarfing" caches found in many commercial multiprocessors. A slightly different hybrid example is the operating system's buffer cache, a section of main memory that is managed by the operating system to store disk blocks. The software that manages the buffer cache operates independently of application software, but the buffer cache can also serve the operating system itself, i.e., the operating system can be a client of the buffer cache, just like application software, and in this case there is no distinction between the manager of the cache and the client of the cache, just like a software-managed cache or a scratch-pad memory.

The reason it is important to identify who or what is responsible for managing the cache contents is that, if given little direct input from the running application, a cache must infer the application's intent, i.e., it must attempt to predict the future behavior of the client (e.g., the application currently executing). Thus, the cache's heuristics for deciding what to retain, when to retain it, where to store it, and when to get rid of it are extremely important. The heuristics for a transparent cache, which are embodied in its organization and various fetch, eviction, and placement strategies, fundamentally determine the cache's effectiveness. By contrast, purely client-managed caches incorporate no cache-based heuristics whatsoever. For example, solid-state caches managed by application software embody no hardware-based heuristics at all; the cache's effectiveness is determined entirely by choices made by the application itself.

This leads us to the three primary components of our framework. Every cache, whether implemented in hardware or software, whether transparent or client managed, whether distributed or monolithic, embodies instances of the following three items:

- The cache's **organization** is the logical arrangement of data stored within its context. For example, an operating system's buffer cache could be organized as an array of queues, with each queue loosely representing the last time a stored object was referenced (e.g., queue1 holds items just arrived, queue2 holds items recently referenced, queue3 holds items not referenced in a while, etc.); a solid-state cache could be organized as an array of sets, each of which contains a handful of cache blocks maintained in a time-ordered manner.
- **Content-management heuristics** represent the decision to cache or not to cache a particular item at a particular time during execution. These can be implemented either at design time by the programmer and/or compiler—or at run time by application software and/or the cache itself. The heuristics can be static in nature, in which case the cache's contents do not change significantly over time, or dynamic in nature, in which case the cache's contents could change radically from moment to moment.
- **Consistency-management heuristics** ensure that the instructions and data that application software expects to receive are the ones the software does, indeed, receive. Like cache contents, consistency can be managed by any of several different

Petitioner Samsung Ex-1009, 0025

entities, including the operating system, the application software, or the cache itself.

These components appear in every cache, and they are present even if not explicitly chosen or implemented. For example, a simple array of blocks is an organization. Even *fetch-nothing* can be considered a simple, if ineffectual, content-management heuristic; and, similarly, having no consistency heuristic at all is still a design choice. At first glance, these mechanisms are all independent; they are largely orthogonal in that the choice made for one component typically does not restrict significantly a designer's available choices for the other components. For instance, one can freely change cache size and associativity simultaneously with the prefetch and write-back policies and also simultaneously with the choice of cache-coherence protocol. Upon deeper inspection, however, many of the issues are heavily interconnected: choices made for one component can make the problems easier or harder to solve for other components. For example, the use of a virtual cache organization places an enormous burden on consistency management that is not present with physical caches (for more detail, see Chapter 4, Section 4.2.1, "Virtual Cache Management).

For further in-depth reading on the topic of transparent processor caches, we suggest as starting points the thorough discussion by Hennessy and Patterson [2003] and Alan J. Smith's excellent survey [1982], which is as pertinent today as it was when it was written (i.e., don't let the publication date fool you). For further reading on both hardware and software techniques for scratch-pad memories specifically and embedded memory systems generally, we suggest as a starting point the inaugural, two-part issue of *ACM Transactions on Embedded Computing Systems* [Jacob & Bhattacharyya 2002].

1.1 Caches, 'Caches,' and "Caches"

Nearly every field imaginable, or at least every field that deals with data in some form or another, has adapted the concept of cache, and every field uses the same term (i.e., cache) to describe what it is that they have built. For example, the digital signal processor (DSP) community uses the word *cache* to describe the dual SRAM scratch-pads found on DSP chips that enable simultaneous access to two data operands [Lapsley et al. 1997], and the high-performance computing community uses the same word to describe the largely autonomous and increasingly complex processor caches that are completely transparent to software and resemble small hardware databases more than anything else. These are both built-in hardware. Software caches are just as prevalent: the world-wide web uses file caching extensively in web proxies [Luotonen & Altis 1994]; network routers cache domain name system (DNS) entries to speed translation from domain names to Internet Protocol (IP) addresses; the operating system stores disk data in its buffer cache (an in-memory cache of disk buffers) to reduce the time spent going to the disk system; and even the disk itself caches buffers in its own local storage.

This is illustrated in Figure 1.1, which shows numerous different caches. The top row presents various configurations of solid-state hardware caches, and the bottom row presents examples of software caches.

- Figure 1.1(a) shows a general-purpose core with its cache. The cache is transparent in that the software executing on the core makes what it believes to be a request to the backing store (the DRAM behind the cache), and the cache intercepts the request. This requires the cache to be aware of its own contents so it can determine if the requested data is stored locally or not.
- Figure 1.1(b) shows another configuration of a general-purpose cache, in which the cache is logically and/or physically split into two separate partitions: one that holds only instructions and one that holds only data. Partitioning caches can improve behavior. For instance, providing a separate bus to each partition potentially doubles the bandwidth to the cache, and devoting one partition to fixed-length instructions can simplify read-out logic by obviating support for variable-sized reads.

Petitioner Samsung Ex-1009, 0026

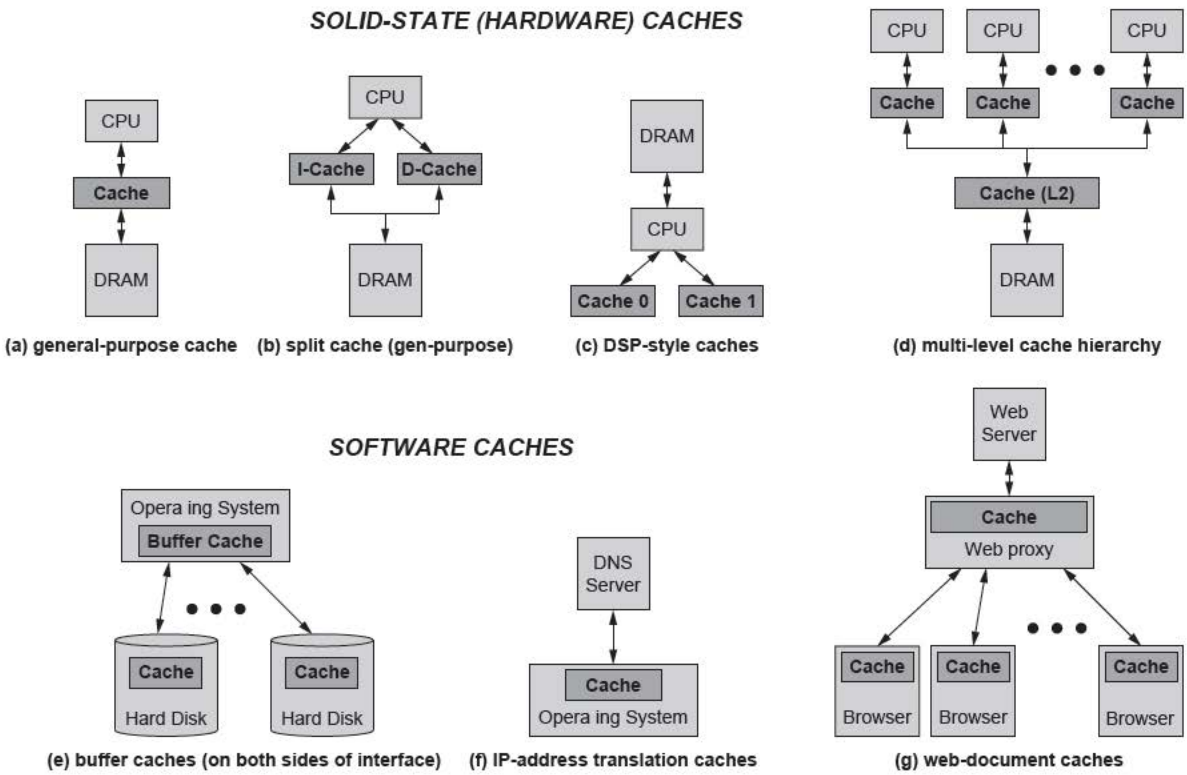


FIGURE 1.1: Examples of caches. The caches are divided into two main groups: solid-state caches (top), and those that are implemented by software mechanisms, typically storing the cached data in main memory (e.g., DRAM) or disk.

- Figure 1.1(c) shows another partitioned configuration, one that is found in most DSP architectures. As we will discuss in more detail later, these caches are an extension of the memory space, and, unlike the previous two organizations, they are not transparent to software. The main item to note is that the two partitions both hold data. This configuration is desirable in DSPs because of the high data bandwidth required by many DSP algorithms (e.g., dot-product calculations such as finite impulse response [FIR] filters), which fetch two different data operands from memory on every cycle.
- Figure 1.1(d) shows a typical cache hierarchy for a multiprocessor organization. Each processor core has its own private cache, and the last-level cache before the backing store (DRAM) is shared by all cores.
- Figure 1.1(e) shows two different types of caches. First, the operating system stores the blocks it has fetched from disk in main memory as part of a *buffer cache*. When an application reads and writes data from/to files on disk, those requests are not serviced by the disk directly but are instead serviced out of the buffer cache. In addition, modern disks implement their own caches as well; these caches are not under the control of the operating system, but instead operate autonomously. We discuss disk caches in more detail in Chapter 22.

Petitioner Samsung Ex-1009, 0027

- Figure 1.1(f) shows another cache maintained by the operating system. When a translation is needed from domain name to IP address, the operating system may or may not actually ask a DNS server directly. Translations may be cached locally to avoid network requests. Similarly, routing information for hosts and subnets is typically cached locally.
- Figure 1.1(g) shows two different types of caches. First, each web browser typically maintains its own local cache of downloaded web documents. These documents are usually stored on the local disk and buffered in main memory as part of the operating system's buffer cache (Figure 1.1(e)). In addition, a request that is sent to a particular web server may or may not actually reach the web server. A web proxy, which behaves like the transparent caches in Figures 1.1(a), (b), and (d), may intercept the request and reply on behalf of the server [Luotonen & Altis 1994].

One of the most important distinctions between different types of cache is their method of addressing. Some caches do not hold a *copy* of a datum; they hold the datum itself. These are often called *scratch-pad memory* and use a separate namespace from the backing store (i.e., the primary memory). A scratch-pad is *non-transparent* in that a program addresses it explicitly. A datum is brought into the scratch-pad by an explicit move that does not destroy the original copy. Therefore, two equal versions of the data remain; there is no attempt by the hardware to keep the versions consistent (to ensure that they always have the same value) because the semantics of the mechanism suggest that the two copies are not, in fact, copies of the same datum, but are instead two independent data. If they are to remain consistent, it is up to software. By contrast, the typical general-purpose cache uses the same namespace as the primary memory system. It is transparent in that a program addresses main memory to access the cache—a program does not explicitly access the cache or even need to know that the cache exists.

How each of these two mechanisms (transparent cache and scratch-pad memory) fits into the microprocessor's memory model is shown in Figure 1.2, using solid-state memories (processor caches and tagless SRAMs) for examples. A general-purpose memory model has a single namespace[1] that is shared by all memory structures. Any datum from any part of the namespace can be found in any of the caches. By contrast, a scratch-pad uses the namespace to directly address the caches. For instance, the DSP memory model shown on the right explicitly places the system's memory structures at specific disjunct locations in the namespace. A particular address corresponds to a particular physical storage device, and unlike a transparently addressed cache, a single address in the scratch-pad's namespace cannot refer to both a cached datum and its copy in the backing store.

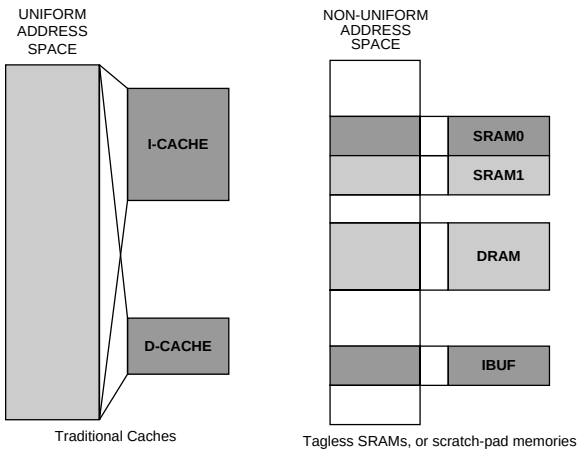


FIGURE 1.2: Transparent caches in a uniform space versus scratch-pad SRAMs in a non-uniform space. Any datum in the memory space can also reside in a cache (thus the designation "transparent"). Only items in certain segments of the memory space can reside in a scratch-pad memory.

[1]Assumes physically indexed caches.

Petitioner Samsung Ex-1009, 0028

As mentioned earlier, what one terms a single cache, or what one terms a set of multiple independent caches, is entirely a matter of convenience and/or need. For instance, the caches in Figure 1.1(b) can be labeled and analyzed as a single cache. One can always analyze the multiple caches at the same "level" in the hierarchy as a single unit (e.g., the L1 caches in Figure 1.1(d), the disk caches in Figure 1.1(e), or the browser caches in Figure 1.1(g)), even though they operate independently. Similarly, one can group the caches in Figures 1.1(d), (e), and (g), which all incorporate entire hierarchies. Depending on the analysis, it may be convenient to treat all levels of cache as a single entity separate from the backing store, but not separate from each other. Going the other way, it is just as correct to treat what appears to be a single cache as a set of independent units. For instance, the caches in Figures 1.1(a) and (f) could each be treated as a collection of independently analyzed blocks.

Finally, embracing all of these different mechanisms as caches, as different aspects of the same thing, enables the solutions developed in different fields of study to cross-pollinate. The goal and point of the formalism, which is presented in detail in the following chapters, are to enable such an analysis. First, the remainder of this chapter presents many of the underlying principles that are common to the caches found across disciplines.

1.2 Locality Principles

Over the decades that the computer has been used and studied, computer engineers have measured program behavior and have made the following observations:

- The memory-access patterns of programs are not random.
- Accesses tend to repeat themselves in time and/or be near each other in the memory address space.

An important note is that these are not guarantees, they are merely observations. The implication of their being observations, as opposed to provably inherent behaviors, is that, unless we can show that new program behaviors will never appear, we need to continue measuring and observing programs and should expect to discover new behaviors occasionally.

As mentioned in the *Overview* chapter, the phenomenon of exhibiting predictable, non-random memory-access behavior is called *locality of reference*. The behavior is so named[2] because we have observed that a program's memory *references* tend to be *localized* in time and space:

- If the program references a datum once, it is likely in the near future to reference that datum again.
- If the program references a datum once, it is likely in the near future to reference nearby data as well.

The first of these phenomena is called *temporal locality*, and the second is called *spatial locality*.

This begs the question: is there nothing else? We have "temporal" and "spatial" aspects of locality. These are brilliantly chosen terms because they imply completeness: time and space cover all possibilities, giving one the distinct if, perhaps, subconscious impression that no other form of locality could possibly exist. It may come as a surprise to some readers that this impression would be wrong.

Remember that these phenomena, these aspects of locality that programs exhibit, are merely observations. They are not inherent to programs, and there is no reason to believe that they are required attributes of a particular program or that they perfectly describe every program that exists. They are merely descriptions of the way that most observed programs tend to behave. Lately, computer engineers have observed a new type of phenomenon.

In the past decade, another class of behavior has sprung up, a consequence of the widespread use of

[2]The term "locality" was put forth by Belady [1966] and was later formalized, in terms of the *working set* model, by Denning [1970].

Petitioner Samsung Ex-1009, 0029

computer graphics algorithms and similar algorithms in other domains, including computer simulation, circuit emulation, interpretation of HDL[3] code, etc. These programs typically walk down dynamic data structures and access the same data in the same order time and time again (e.g., walking through the components of a 3D scene each time a new video frame is created), so their behavior is inherently predictable. However, the time between two successive accesses to the same data is large (thus, the program exhibits no significant temporal locality), and the data objects are generally nowhere near each other in the memory space (thus, the program exhibits no significant spatial locality). According to our existing definitions of locality, these programs exhibit no significant locality,[4] but, nonetheless, the programs of this type exhibit regular, predictable, exploitable behavior. It is hard to categorize this type of behavior because, between time and space, we seem to have everything covered. So it is easy to overlook the fact that this behavior even exists. But exist it clearly does, even if we have no term for it and even if our naming continuum seems to allow no room for a new term to exist.

For lack of a better term, we will call this behavior *algorithmic locality*.

The following sections describe these phenomena in more detail.

1.2.1 Temporal Locality

Temporal locality is the tendency of programs to use data items over and again during the course of their execution. This is the founding principle behind caches and gives a clear guide to an appropriate data-management heuristic. If a program uses an instruction or data variable, it is probably a good idea to keep that instruction or data item nearby in case the program wants it again in the near future.

One corresponding data-management policy or heuristic that exploits this type of locality is *demand-fetch*. In this policy, when the program *demands* (references) an instruction or data item, the cache hardware or software *fetches* the item from memory and retains it in the cache. The policy is easily implemented in hardware. As a side effect of bringing any data (either instructions or program variables and constants) into the processor core from memory, a copy of that data is stored into the cache. Before looking to memory for a particular data item, the cache is searched for a local copy.

The only real limitation in exploiting this form of locality is cache storage size. If the cache is large enough, no data item ever need be fetched from the backing store or written to the backing store more than once. Note that the *backing store* is the term for the level of storage beneath the cache, whether that level is main memory, the disk system, or another level of cache. The term can also be used to mean, collectively, the remainder of the memory hierarchy below the cache level in question.

1.2.2 Spatial Locality

Spatial locality arises because of the tendency of programmers and compilers to cluster related objects together in the memory space. As a result of this, and also because of the human-like way that programs tend to work on related data items one right after the other, memory references within a narrow range of time tend to be clustered together in the memory space. The classic example of this behavior is array processing, in which the elements of an array are processed one right after the other: elements *i* and *i+1* are adjacent in the memory space, and element *i+1* is usually processed immediately after element *i*.

This behavior, like temporal locality, also points to an obvious data-management policy, typically called *lookahead*: when instruction *i* or data item *i* is requested by the program, that datum should be brought into the processor core and, in addition,

[3]HDLs are *Hardware Description Languages* such as Verilog or VHDL, which are commonly used in the specification and design of computer circuits and systems.

[4]Another way of looking at this is that the programs exhibit temporal and/or spatial locality on scales so large that the terms become meaningless.

Petitioner Samsung Ex-1009, 0030

instruction or data item *i+1* should be brought in as well. The simplest form of this policy is to choose judiciously the data-access granularity: to build caches in atomic units called *cache blocks* that are larger than a single data item. For example, a processor in which the fundamental data unit is a 4-byte word might use a 16-byte cache block containing 4 data words, or a 32-byte cache block containing 8 data words, and so on. Because the cache block is *atomic* (meaning that this is the granularity of data access between the cache and memory—the amount of data that is moved for every[5] read or write), a program's request for a single data item will result in the surrounding data items all being fetched from memory as a matter of course.

The use of a cache block that is larger than a single data word is very effective, leads to a simple implementation, and can be thought of as a *passive* form of exploiting spatial locality. Another very effective way to exploit spatial locality is *prefetching*, which, by comparison, is an *active* form. Prefetching (which can be done in hardware, software, or a combination of the two) is a mechanism in which the program's request for an instruction or data item causes more data than just the requested cache block to be fetched from memory. An algorithm uses the memory address with the associated recent history of requested addresses to speculate on which cache block or cache blocks might be used next. An obvious hardware algorithm would be something like fetch-next-block, called *one-block lookahead* prefetching [Smith 1982], which can be seen as a simple extension of the cache block granularity of access. An obvious software algorithm would be something like fetch-next-array-element, which cannot be performed in hardware alone because it requires information about the size of each data element. Much more sophisticated (and complex) algorithms have been explored as well.

The limit on exploiting this type of locality is due to program behavior and the algorithm's accuracy of prediction. If the program tends to access instructions and data in a largely sequential manner, then a simple mechanism is all that is necessary, and the performance results can be extremely good. If the program behavior is more complex (say, for example, the program jumps around from cache block to cache block, but in a predictable manner), a static algorithm will not be capable of adapting and will therefore do poorly. In this case, a more sophisticated prefetching algorithm might be able to track program behavior and correctly predict which block to fetch next.

1.2.3 Algorithmic Locality

The last discussion leads us into algorithmic locality. Algorithmic locality arises when a program repeatedly accesses data items or executes code blocks that are distributed widely across the memory space. The net result is program behavior that tends to be predictable, but, nonetheless, cannot be captured by using granularity of access or simple lookahead-prefetching mechanisms because the data items referenced by the program are nowhere near each other. Similarly, this behavior cannot be captured by simply increasing the cache size by a factor of two or three because the data and code sizes involved are typically very large.

Many applications that exhibit this aspect of locality perform repeated operations on very large data sets that are stored in dynamic data structures that do not change shape radically over short periods of time. Example applications of this sort include graphics processing, computer simulation, circuit emulation, and various forms of HDL processing (interpretation, synthesis, validation, verification, rule-checking, regression testing, etc.).

Many of the steps in 3D computer-graphics processing find the program repeatedly traversing a large dynamic data structure representing the shape of the physical object being rendered. One example is the *Z-buffer algorithm*, in which the depth of each object

[5]However, there are always exceptions (see "Other Policies and Strategies for Transparent (Processor) Caches" in Section 3.1).

Petitioner Samsung Ex-1009, 0031

in the 3D scene is used to paint the two-dimensional (2D) image of the scene:

```
for each polygon P in scene {
   for each pixel p in P {
      if (depth of p at i,j < image[i,j].depth) {
         image[i,j].color = p.color
         image[i,j].depth = depth of p at i,j
      }
   }
}
```

The algorithm walks through a list of polygons, performing a number of calculations on each polygon. As a scene is portrayed using an increasingly fine mesh of polygons, the more realistic the rendered scene tends to look. Thus, in practice, the number of polygons tends to be large, and the number of pixels per polygon tends to be small.

This algorithm will walk through the polygon list over and over for each frame of a video sequence. The polygon list is typically a dynamic data structure such as a linked list, not a static array. Because there can be no guarantee about the physical placement in memory for any particular element in the list, one cannot expect to see this algorithm exhibit significant spatial locality. Because the polygon list is large, significant time will pass before any polygon is revisited by the algorithm, and thus, the algorithm will not exhibit significant temporal locality either. The algorithm is highly predictable in terms of its memory-access patterns. It is largely because of this predictability that graphics coprocessors have been able to make such an enormous impact on the computing industry.

Because the predictability is exploitable, it most certainly represents a type of locality, and it arises from the behavior of the algorithm (thus, the name). To exploit this locality, one must know the algorithm's behavior, which means the optimization could be a run-time mechanism that observes program behavior and adapts to it, or it could be a relatively simple prefetching code inserted by the compiler or programmer at design time. For instance, the algorithm could be rewritten as follows to prefetch polygons into the processor core before they are actually needed for processing (the primary difference is in bold and assumes some form of linked-list representation for polygons):

```
for each polygon P in scene {
   prefetch P.next
   for each pixel p in P {
      if (depth of p at i,j < image[i,j].depth) {
         image[i,j].color = p.color
         image[i,j].depth = depth of p at i,j
      }
   }
}
```

This is but a simple, obvious example (for more discussion on prefetching, see Chapter 3, Section 3.1.2, "On-Line Prefetching Heuristics," and Section 3.2.2, "Off-Line Prefetching Heuristics"). Exploiting this type of locality is limited only by the imagination of the algorithm's writers and/or the hardware's designers. For instance, more sophisticated software algorithms can prefetch more than one polygon ahead (i.e., further out into the program's "future" execution), and larger prefetch buffers can accommodate more data fetched. Both would allow the algorithm to tolerate very long memory latencies.

1.2.4 Geographical Locality? Demographical Locality?

As we will discuss in more detail later, designers continue to refine cache performance (as well as power dissipation, timeliness, predictability, etc.), often by the divide-and-conquer approach of classification, to wit:

1. Identify classes of application/algorithm behavior or data types that exhibit recognizable behavior.
2. Exploit the (predictable) behaviors that are exhibited.

This applies very well to locality principles: demand-fetch caches exploit temporal locality; large block sizes and/or simple prefetching schemes exploit spatial locality; and software-directed prefetching schemes and hardware prefetching schemes that use dynamic pattern recognition exploit algorithmic locality. As new applications emerge that can benefit

Petitioner Samsung Ex-1009, 0032

from caching, new forms of behavior are seen, and new mechanisms to exploit that behavior are used. Web caching (and content delivery in the general sense) is but one example that introduces radically different new forms of behavior; entire geographical regions and/or demographical subsets within those regions can exhibit recognizable, exploitable behavior. Consider, for example, a prefetching heuristic for proxy caches that would improve North American network traffic and browser latency on Superbowl Sunday. Attempting to think about such problems without considering the cause of the behavior, e.g., trying to optimize a content-delivery system by treating data traffic in terms of only temporal and spatial locality and ignoring the human element behind the traffic, is guaranteed to produce a second-rate result.

1.3 What to Cache, Where to Put It, and How to Maintain It

As illustrated in Figure 1.3, there are five orthogonal components comprising a cache design: the decision of what to put into the cache, a content-management heuristic that can be made either dynamically or statically by the programmer, by the compiler, by the running application, by the operating system, or by the cache itself; the decision of where and how to store

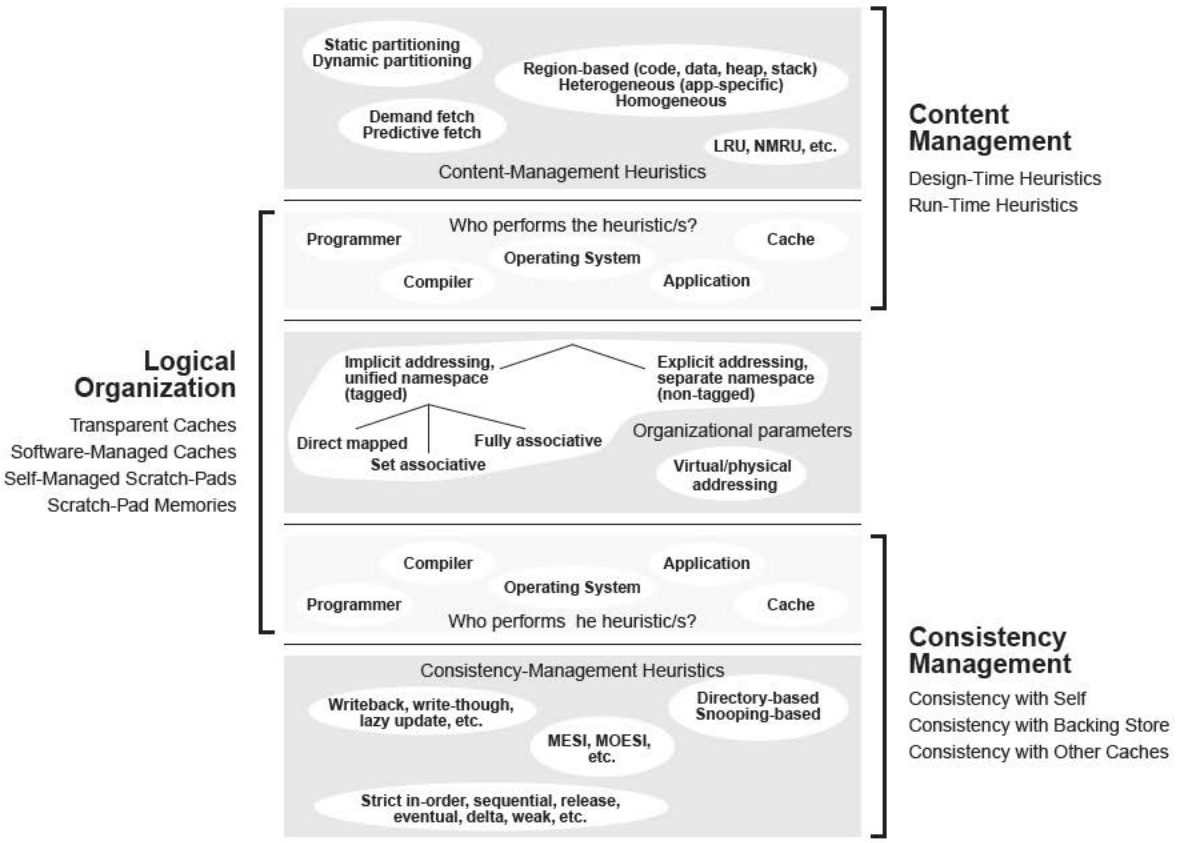


FIGURE 1.3: The orthogonal design choices that make up a cache. Any cache design, whether software based or entirely solid state, comprises a logical storage organization, one or more content-management heuristics, and one or more consistency-management heuristics.

Petitioner Samsung Ex-1009, 0033

cached data within the cache's physical extent and/or logical arrangement (its logical organization); and the mechanism by which the cache (together with any executing software) ensures the consistency of data stored within it, a heuristic that can be performed by any of several entities, including the application code, the operating system, or the cache itself. A cache can be built by combining essentially any logical organization with essentially any set of management heuristics.

This section sets up the next three chapters of the book. The three primary issues illustrated in Figure 1.3 (Logical Organization, Content Management, and Consistency Management) will be discussed here in an overview fashion, and the next three chapters will go into each in more detail.

1.3.1 Logical Organization Basics: Blocks, Tags, Sets

A cache stores chunks of data (called *cache blocks* or *cache lines*) that come from the backing store. A consequence of this block-oriented arrangement is that we logically divide the backing store into equal, cache-block-sized chunks. A useful side effect of this is the provision for a simple and effective means to identify a particular block: the *block ID*. A subset of the address, the block ID is illustrated in Figure 1.4, which shows a 32-bit byte main memory address as a bit vector divided into two components: the cache block ID and the byte's offset within that cache block. Note that the "byte in block" component is given as 4 bits. This example, therefore, represents a cache block of size 16 bytes. If a cache were to contain disk blocks, with several disk blocks to a cache block, then this arrangement would be the same, except that the 32-bit physical memory address would become a disk block number.

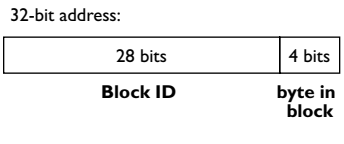


FIGURE 1.4: A 32-bit address.

Because a cache is typically much smaller than the backing store, there is a good possibility that any particular requested datum is not in the cache. Therefore, some mechanism must indicate whether any particular datum is present in the cache or not. The *cache tags* fill this purpose. The tags, a set of block IDs, comprise a list of valid entries in the cache, with one tag per data entry. The basic structure is illustrated in Figure 1.5, which shows a cache as a set of cache entries [Smith 1982]. This can take any form: a solid-state array (e.g., an SRAM implementation) or, if the cache is implemented in software, a binary search tree or even a short linked list.

If a cache's size is but one block, then its organization should be relatively obvious: data goes into the one block, displacing whatever was previously there. However, as soon as a cache has more than one block, its organization becomes an issue: where should that second block be placed? The data corresponding to a particular block ID can only be found in one cache entry at a time, but there is still flexibility in the choice of which entry in the cache to choose. We will use the vertical and horizontal dimensions in our figures to denote a group of equivalence classes; each row is an equivalence class called a cache *set*. The single, unique set to which a particular chunk of

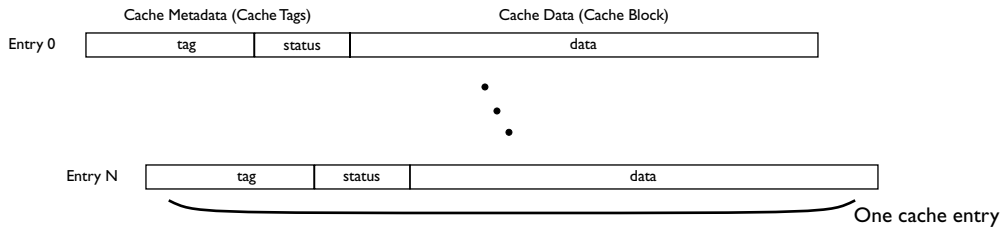


FIGURE 1.5: Basic cache structure. A cache is composed of two main parts: The cache metadata (or, simply, tags) and the cache data. Each data entry is termed a *cache line* or *cache block*. The tag entries identify the contents of their corresponding data entries. Status information includes a valid bit.

Petitioner Samsung Ex-1009, 0034

data belongs is typically determined by a subset of bits taken from its block ID, as shown in Figure 1.6. Once the set is determined, the data chunk may be placed within any block in that set.

There are three basic cache organizations that arise from this equivalence-class mechanism: *direct-mapped*, *fully associative*, and *set associative*. Direct-mapped caches have sets with only one block in each set; each block is its own equivalence class. Fully associative caches (also called *content-addressable memories*) have only one set that encompasses all blocks; thus, all blocks in the cache are in the same equivalence class. Set-associative caches strike a balance between these two choices. A set-associate cache has more than one set (if only one, the cache would be fully associative), and each set in the cache incorporates more than one block in its equivalence class (if only one, the cache would be direct-mapped). Figure 1.7 illustrates the three cache organizations by demonstrating the various ways a cache of eight blocks can be organized.

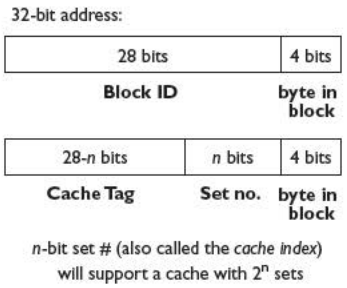


FIGURE 1.6: Different address views.

1.3.2 Content Management: To Cache or Not to Cache

As an application executes over time, it makes a sequence of references to memory (or, more generally, the backing store), each of which is a potential candidate for caching. The purpose of a content-management solution is to determine which references are the best candidates for caching and to ensure that their corresponding data values are in the cache at the time the application requests them. Simultaneously, it must consider the currently cached items and decide which should be evicted to make room for more important data not yet cached. The optimal algorithm is one that can look into the future and determine what will be needed and ensure that all important data words, up to the limits of the cache's capacity and organization, are in the cache when requested. This is similar in spirit to Belady's MIN algorithm for disk block replacement [Belady 1966]. Both schemes rely upon an oracle to guarantee optimal performance.

Given the absence of an oracle, one is left with sub-optimal approaches, i.e., heuristics that can, at best, approximate the optimal approach, making do with imperfect and/or incomplete information. These heuristics take as input anything that might describe a data item: e.g., who is using it, how it is being used, how important it is, and at what point in the application's execution is the item being considered. Using this, the heuristic makes a "yes/no" decision to cache the item. Note that the decision can be made at any of several different times, by any of several different entities. The time can be relatively early on in the process—for instance, when the programmer writes and compiles the application code—or

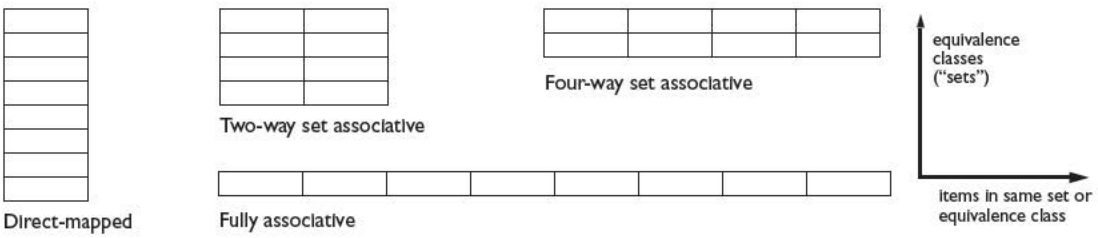


FIGURE 1.7: Four organizations for a cache of eight blocks. Eight blocks can be configured in a direct-mapped organization, a fully associative organization, or any of two different set-associative organizations.

Petitioner Samsung Ex-1009, 0035

it can be made relatively late in the process—for instance, while the application code is executing. The decision can be made by the programmer, the compiler, the operating system, the application code, or the cache itself. In addition, some decisions will be inherently *predictive* in nature (e.g., a design-time heuristic embedded in the compiler determines that a particular reference should hit the cache, so proactive steps such as prefetching should be performed to ensure that the data is in the cache at the time the reference is made), while other decisions will be inherently *reactive* in nature (e.g., a run-time heuristic in the cache determines that a requested item is valuable and therefore should remain in the cache once fetched from the backing store).

An important thing to bear in mind when considering the role of the content-management solution (i.e., the set of design-time and/or run-time heuristics) is that the logical organization of the cache has the final word on whether something a heuristic decides to cache will, in fact, remain cached. The logical organization defines how and where cached data is stored, and if a heuristic chooses to cache two items that the logical organization cannot hold simultaneously, then one of the two must go. For example, consider two back-to-back references made to items deemed important by some heuristic, but positioned in the backing store's namespace such that they are assigned to the same set in the cache (this would happen if the two items were stored in memory exactly 1 MB apart, given a cache of 1 MB in size or less), and, further, assume that the cache is not set associative (each set can hold only one data item). The cache can hold only one of these items, and so the second reference would displace the first, irrespective of the heuristic's wishes.

1.3.3 Consistency Management: Its Responsibilities

Consistency management comprises three main charges:

1. Keep the cache consistent with itself
2. Keep the cache consistent with the backing store
3. Keep the cache consistent with other caches

In many cases, these goals can be accomplished simply. In most cases, the choice of logical organization can complicate things tremendously.

Keep the Cache Consistent with Itself

There should never be two copies of a single item in different places of the cache, unless it is guaranteed that they will always have the same value. In general, this tends not to be a problem, provided the cache is physically indexed and physically tagged.

Many operating systems allow two completely unrelated virtual addresses to map to the same physical address, creating a *synonym*. The synonym problem is non-trivial and is discussed in more detail in Chapter 4. It creates a problem by allowing the physical datum to lie in two different virtually indexed cache sets at the same time, immediately creating an inconsistency as soon as one of those copies is written. If the cache's sets are referenced using a physical address (i.e., an address corresponding to the backing store, which would uniquely identify the datum), then there is no problem.

Using tags that correspond to the backing store solves another problem. Even if some mechanism ensures that two synonyms map to the same set in the cache, the cache could nonetheless hold the two different copies if the cache is set associative and the two cache blocks are not tagged with an ID taken from the backing store. For example, if the cache uses virtual tags or if the cache fails to ensure that the blocks held in a set all have unique tags, then an inconsistency could occur.

Keep the Cache Consistent with the Backing Store

The value kept in the backing store should be up-to-date with any changes made to the version stored in the cache, but only to the point that it is possible to ensure that any read requests to the version in the backing store (e.g., from another processor) return the most recently written value stored in the cache.

Two typical mechanisms used in general-purpose caches to effect this responsibility are *write-back* and *write-through* policies. When writing to a cache, the backing store is implicitly updated, but when? Two

Petitioner Samsung Ex-1009, 0036

obvious choices are *immediately* and *later*; the first is write-through, and the second is write back. Write-through offers the advantage of having a very small window in which the backing store has an outdated ("stale") copy of a datum. Write-back takes advantage of locality in that, if a block is written, the block will likely be written again in the near future, in which case it would make no sense to transmit both values to the backing store. A write-back cache keeps track of blocks that have been written (called "dirty") and writes their contents to memory only upon block replacement, potentially saving significant bandwidth to the backing store.

The disadvantage of write-through is its cost: shoveling every write to the cache into the backing store at the same time becomes expensive, because the backing store typically cannot provide the same bandwidth as the cache. The solution is to provide an additional fast memory of some sort that is physically part of the cache, but logically part of the backing store. Popular choices for this fast memory are FIFO (first-in first-out) write buffers, coalescing write buffers, or searchable write buffers, a.k.a. write caches. The point of this hardware is to decouple the bursty nature of the microprocessor's write operations from the backing store and to dribble the written data to the backing store at whatever speed it can handle. The simple write buffer must be drained if it fills or if a cache miss occurs. More complex write buffers are capable of much more sophisticated behavior.

The primary disadvantage of write-back caches is their interaction with other *processors* and other *processes*. In the first case, in a multiprocessor organization, having a long window of time wherein the backing store can have the wrong value for a particular datum increases the likelihood that another processor sharing the backing store will read that (wrong) datum. We will discuss this later in Chapter 4. In the second case, the virtual memory system's synonym problem complicates things tremendously. Consider, for example, the case when a process writes into its address space, exits, and part of its data space is reused when another process is created or dynamically allocates memory in its heap or stack segments. Nothing prevents a write-back virtually indexed cache from overwriting the new process's address space when those cache blocks are finally replaced. Even if the operating system wipes the data pages clean, the old written data from the first process can still be found in the cache if the operating system uses different virtual addresses or a different address-space identifier than the original process.

Keep the Cache Consistent with Other Caches

Other caches in the system should all be treated much like the backing store in terms of keeping the various copies of a datum up-to-date, as in Item #2 above.

Note that we are using the term *consistent* here in the normal, Merriam-Webster sense; we are not talking about *memory consistency*, a mechanism to specify the ordering of reads and writes that a (multiprocessor) system as a whole sees.

1.3.4 Inclusion and Exclusion

Before we move on, an aspect of cache organization and operation needs discussion: inclusion. The concept of *inclusion* combines all three aspects of caching just described. Inclusion can be defined by a cache's logical organization, and it is enforced by the cache's content- and consistency-management policies.

Figure 1.8 gives a picture of the canonical memory hierarchy. The hierarchy is *vertically partitioned* into separate storage levels, and each level can be *horizontally partitioned* further [Lee & Tyson 2000]. This horizontal partitioning is also called a *multi-lateral cache* [Rivers et al. 1997]. Partitioning is a powerful mechanism. For instance, Lee and Tyson [2000] note that much work in cache-level energy reduction has been to exploit these divisions. One can use vertical partitioning and move more frequently accessed data into storage units that are closer to the microprocessor—units that can be made significantly smaller than units further out from the microprocessor (e.g., the line buffer [Wilson et al. 1996]), and thus typically consume less energy per reference [Kin et al. 1997, 2000]. Similarly, one can use horizontal partitioning and slice a logically monolithic cache into smaller

Petitioner Samsung Ex-1009, 0037

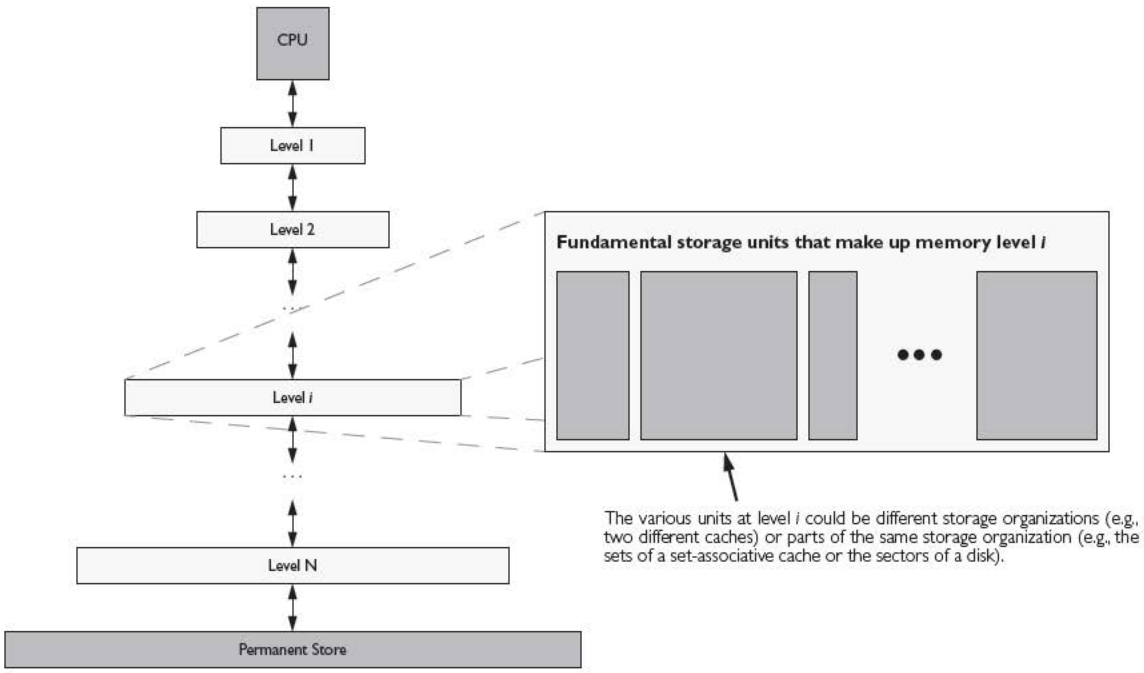


FIGURE 1.8: A canonical memory hierarchy. As one moves down through the hierarchy, from the central processor toward the permanent store, the levels are larger but slower. At each level, the storage may be a monolithic organization, or it may be partitioned into multiple segments we have called generically "units." Every fundamental unit has an exclusive or inclusive relationship with every other unit in the hierarchy.

segments (e.g., subbanks [Ghose & Kamble 1999]), each of which consumes a fraction of the whole cache's energy and is only driven when a reference targets it directly.

The principles of *inclusion* and *exclusion* define a particular class of relationship that exists between any two partitions in a cache system. Whether the two partitions or "units" are found at the same level in the hierarchy or at different levels in the hierarchy, there is a relationship that defines what the expected intersection of those two units would produce. That relationship is either exclusive or inclusive (or a hybrid of the two). Figure 1.9 illustrates a specific memory hierarchy with many of its relationships indicated.

An inclusive relationship between two fundamental units is one in which every cached item found in one of the units has a copy found in the other. For example, this is the relationship that is found in many (but not all) general-purpose, processor cache hierarchies. Every cache level maintains an inclusive relationship with the cache level immediately beneath it, and the lowest level cache maintains an inclusive relationship with main memory. When data is moved from main memory into the cache hierarchy, a copy of it remains in main memory, and this copy is kept *consistent* with the copy in the caches; i.e., when the cached copy is modified, those modifications are propagated at some point in time to main memory. Similarly, copies of a datum held within lower levels of the cache hierarchy are kept consistent with the copies in the higher cache levels. All this is done transparently, without any explicit direction by the cache system's client.

Keeping all of these additional copies spread around the memory system would seem wasteful. Why is it done? It is done for simplicity of design and

Petitioner Samsung Ex-1009, 0038

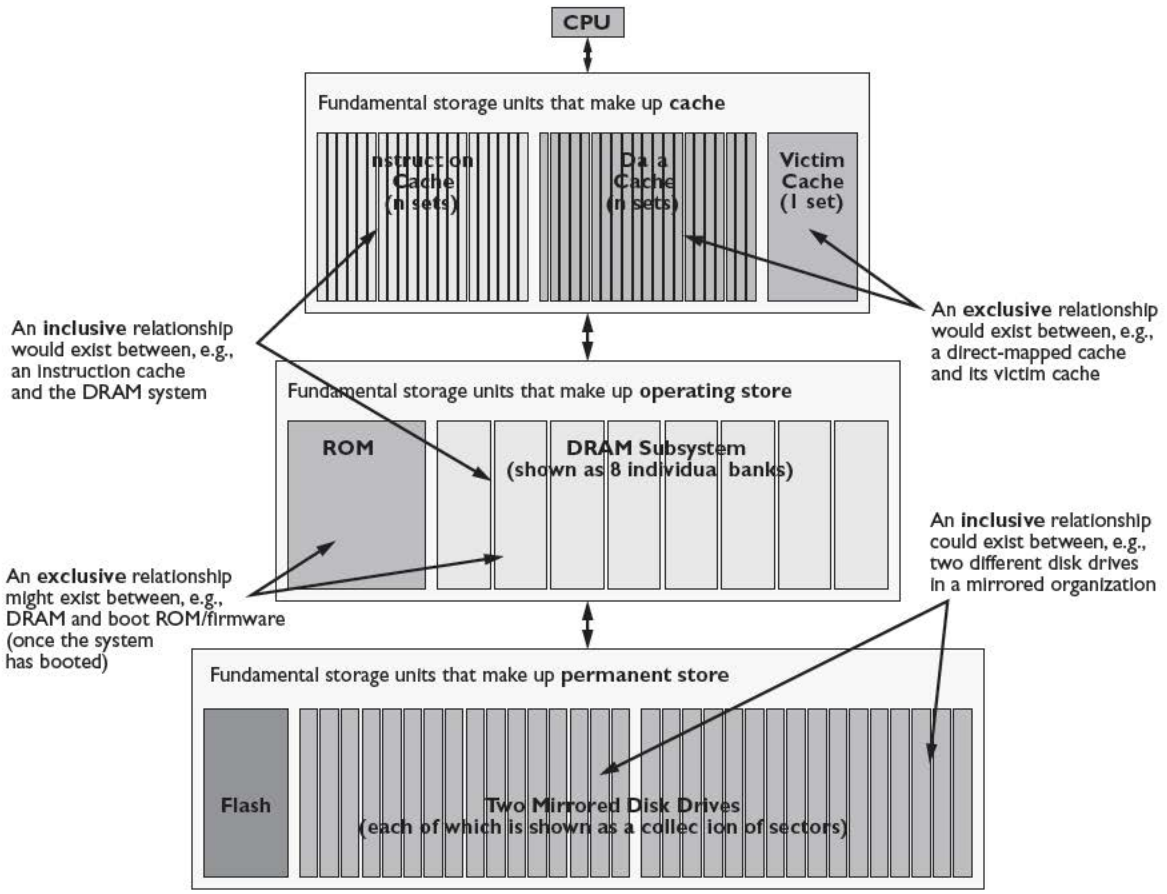


FIGURE 1.9: A specific memory hierarchy. A specific example of a memory system shows various vertical and horizontal partitions that can exist, as well as the inclusive/exclusive relationship between many of the fundamental storage units.

correctness of design. If an inclusive relationship exists between partitions A and B (everything in A is also held in B, and B is kept consistent with the copy in A), then A can be considered nothing more than an optimized lookup: the contents of A could be lost forever, and the only significant effect on the system would be a momentary lapse of performance, a slightly higher energy drain, until A is refilled from B with the data it lost. This is not a hypothetical situation. Low-power caching strategies exist that temporarily power-down portions of the cache when the cache seems underutilized (e.g., a bank at a time). Similarly, the arrangement simplifies eviction: a block can be discarded from a cache at any time if it is known that the next level in the hierarchy has a consistent copy.

An exclusive relationship between two units is one in which the expected intersection between those units is the null set: an exclusive relationship between A and B specifies that any given item is either in A or B or in neither, but it absolutely should not be in both. The content- and consistency-management protocols, whether implemented by the application software, the operating system, the cache, or some other entity, are responsible for maintaining

Petitioner Samsung Ex-1009, 0039

exclusion between the two partitions. In particular, in many designs, when an item is moved from A to B, the item must be swapped with another item on the other side (which is moved from B to A). This is required because, unlike inclusive relationships in which a replaced item can be simply and safely discarded, there are no "copies" of data in an exclusive relationship. All instances of data are, in a sense, *originals*—loss of a data item is not recoverable; it is an error situation.

Note that there are many examples of cache units in which this swapping of data never happens because data items are never moved between units. For example, the sets of a direct-mapped or set-associative cache exhibit an exclusive relationship with each other; they are partitions of a cache level, and when data is brought into that cache level, the data goes to the one partition to which it is assigned, and it never gets placed into any other set, ever.[6]

1.4 Insights and Optimizations

1.4.1 Perspective

Using general terms, we intentionally describe traditional concepts such as locality, inclusion/exclusion, coherence/consistency, cache organization, protocols, policies, heuristics, etc. to make it clear that they are universally applicable to all manners of cache implementation. One item that should follow from this general treatment is for the reader to recognize potential relationships between mechanisms that are not usually considered related or even compatible. This general treatment should make obvious to the reader potentially interesting avenues for research. As an example,

- Figure 1.3 makes it clear that having different entities perform the same type of heuristic is a complementary action. For instance, both hardware and software could perform content management simultaneously, much in line with hybrid branch predictors or as is already done with software-prefetching schemes (where both cache and software can decide to store items in the cache).
- Considering the sets of a transparent cache as having an exclusive relationship with each other begs the question, *why?* Why not allow some things to reside in multiple sets of a direct-mapped or set-associative cache? Perhaps naming can be a way to accomplish this simply, i.e., through the compiler or virtual memory system (using a virtually indexed cache). Note that this would complicate consistency management.
- Similarly, if one looks at a (set-associative) cache as just a collection of equivalence classes, there is no reason to require each set to have the same cardinality, i.e., to have the same degree of set associativity. Some sets could be large, and others could be small, just as a cache partition could be large or small. Some could have different organizations entirely, with naming schemes used to assign items to different sets or equivalence classes. This is already done in file caching (e.g., see Chapter 2, Section 2.6.2, "A Software Implementation: BSD's Buffer Cache"), and in a solid-state implementation it would be similar in spirit to the work done by Farrens, Rivers, Tyson, and others [Tyson et al. 1995, Rivers & Davidson 1996, Johnson & Hwu 1997] in their work on alternative content-management strategies (discussed in more detail in Chapter 3, Section 3.1.1, "On-Line Partitioning Heuristics").
- Given that content management is orthogonal to cache organization (clear from Figure 1.3), one can start to combine traditionally separated heuristics and organizations. For instance, one could use scratch-pad placement solutions to solve the problems of cache-set assignment.

[6]This is mostly, but not completely, true. See the discussion of problems in virtual caches (Chapter 4, Section 4.2.1, "Virtual Cache Management"), and the section on trace caches (Chapter 2, Section 2.6.3, "Another Dynamic Cache Block: Trace Caches").

Petitioner Samsung Ex-1009, 0040

The list goes on. The whole point of starting from the basics and using general terms is to facilitate novel applications of the tremendous amount of work done in cache design (across many different areas) into new areas.

Reconsidering Bandwidth and Block Size: DSP Caches

One example of the types of opportunities that become available when traditional lines are crossed is a study originally done several years ago [Srinivasan et al. 2001], the idea for which only came about because the traditional line between DSP caches and general-purpose caches had been crossed by the DSP community in their desire to compile C code for DSP architectures. The following is a brief excerpt[7] of the study.

As mentioned earlier, DSPs typically use a non-uniform addressing model in which the primary components of the memory system, the DRAM and dual tagless SRAMs, are referenced through separate segments of the address space (see Figure 1.2). The recent trend of programming DSPs in high-level languages instead of assembly code has exposed this memory model as a potential weakness, as the model makes for a poor compiler target. In many of today's high-performance DSPs, this non-uniform model is being replaced by a uniform model, a transparent organization like that of most general-purpose systems, in which all memory structures share the same address space as the DRAM system.

This opens up an interesting design space to study—how best to build transparent caches for DSPs, perhaps exploiting the regular behavior exhibited by many applications on DSP processors (similar in vein to vector apps on vector processors), as well as their use of memory resources (ports), a behavior that resembles an exclusive partition and is very predictable. For example, DSPs might be able to make good use of large cache blocks, particularly block sizes where a general-purpose application might exhibit high degrees of cache pollution. An obvious downside of doing so is the long fill-time for large blocks, but this can be offset with increased memory bandwidth.

We look at two different organizations illustrated in Figure 1.10. The first is an extremely simple organization: two blocks, each one dedicated to a different load/store unit. The second adds a single-block *victim cache* [Jouppi 1990] to each of the blocks in the first organization. In this organization, each victim cache block holds only victims taken from its associated cache block. Note that in all organizations, cache reads may be satisfied out of any block in the cache. The tying of a block to a particular load/store unit is simply used in the replacement policy; it does not affect the cache lookup mechanism. Any load/store unit can read a value from any block in the cache, which makes the cache effectively fully associative.

This is a rather non-traditional cache organization, but it follows from our observations of the behavior of 'C6000 DSP applications. In particular, we found that the C compiler tends to assign load and store instructions to particular ports in a manner that does not change dynamically: to wit, if the DSP reads a particular datum from a particular port, the DSP is less than 50% likely to read that datum from the other port in the near future, and it is *far* less than 50% likely to *write* that datum to the other port in the near future. We exploit the observed behavior by constraining which blocks can be replaced and written from which ports, thereby creating a simpler cache. Note that this behavior is specific to compiled applications, as opposed to DSP applications written by hand in assembly code, but it is also likely that the observed behavior would be seen in any DSP with a unified memory space and a high-level language compiler.

Figure 1.11 shows the performance of the two organizations, compared to the performance of the 64-KB scratch-pad SRAM at 400 MB/s. Note that the vertical scales of the various graphs differ.

[7]This work is based on an earlier work: "Transparent data-memory organizations for digital signal processors," by S. Srinivasan, V. Cuppu, and B. Jacob, in *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2001)*, November 2001. © ACM, 2001. http://doi.acm.org/10.1145/502217.502224

Petitioner Samsung Ex-1009, 0041

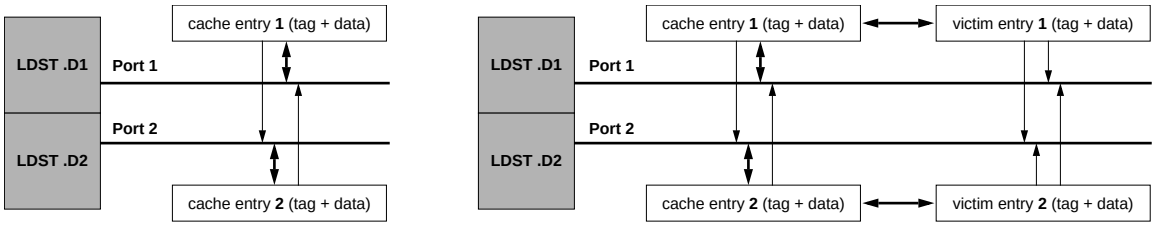


FIGURE 1.10: Slightly non-traditional cache organizations. Each load/store unit can read from any block in the cache, but it can only write to the primary block that is assigned to it. Similarly, load misses only fill the cache block assigned to a load/store unit. The victim cache organization adds two blocks to the cache that are only filled by blocks replaced from the corresponding primary blocks.

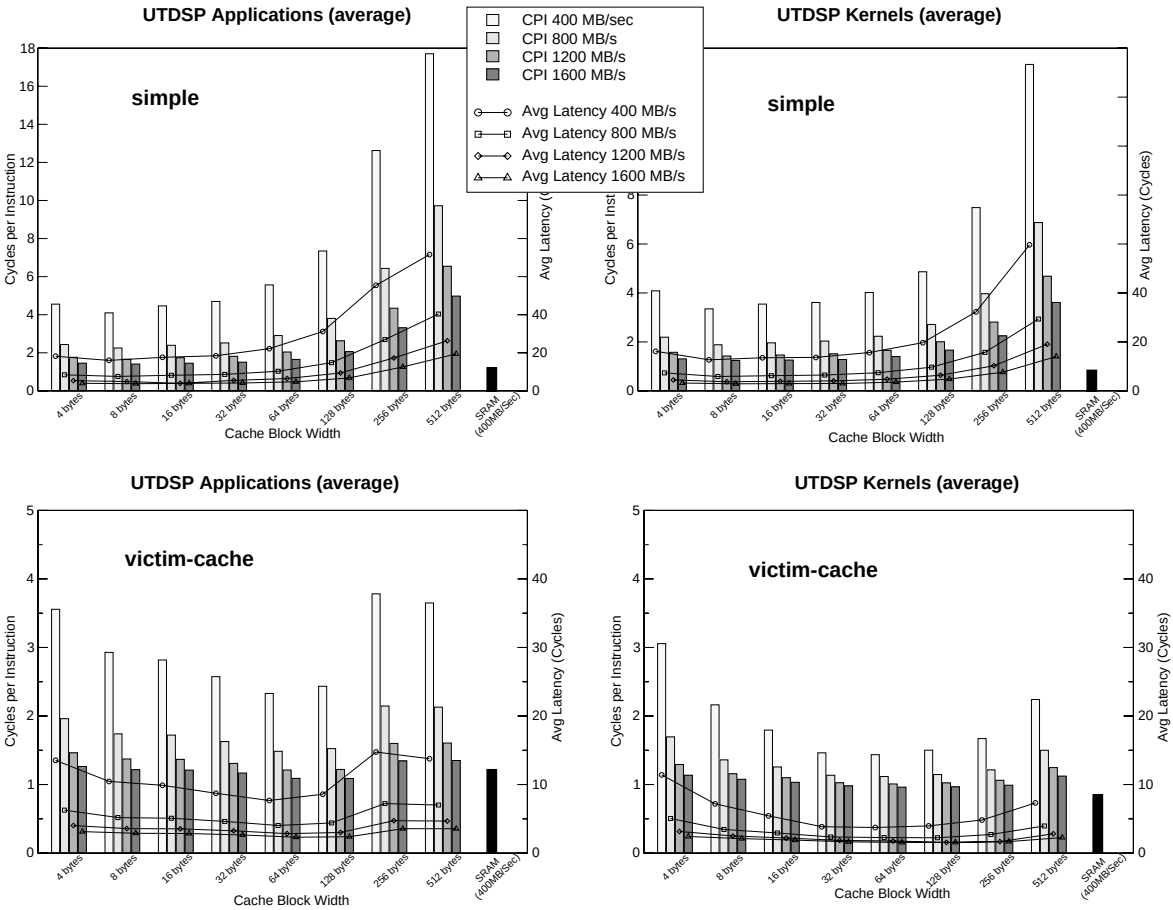


FIGURE 1.11: Performance graphs. The top two graphs give results for the "simple" organization. The bottom two give results for the "victim cache" organization.

Petitioner Samsung Ex-1009, 0042

Execution time (in CPI) is given on the left side of each graph, corresponding to the histograms; average latency per memory instruction is given on the right side of each graph, corresponding to the line graphs. Memory bandwidth and cache block size are the variables (note that total cache capacity is not kept constant and instead grows as block size grows).

In general, if one is willing to pay for more bandwidth, one can get away with significantly less on-chip storage and reach the same levels of performance. This represents an interesting trade-off between chip costs in on-chip storage and system costs in memory bandwidth. That one can approach the performance of the traditional DSP memory organization with such a simple structure is a natural consequence of the fact that DSP applications typically stream data, obviating large cache structures and making stream buffers more attractive.

Reconsidering Permanent Store: Plan 9's File System

Another example of interesting cache designs is Plan 9's file system,[8] in which the designers re-defined the concept of "permanent store" [Quinlan 1991]. The operating system treats the entire magnetic disk system as a mere cache for the backup system (the WORM [write once, read many] jukebox, a jukebox of write-once/read-many optical disks[9]), which is considered the repository of the true file system and thus the ultimate in permanent backing store.

The work is based upon earlier explorations in the same general direction. For instance, Garfinkel [1986] describes a file-system API (application programming interface) that moves copies of files to backup whenever those files are written. The File Motel [Hume 1988] uses the WORM drive as a transparent backup for the file system, which resides as normal on the magnetic disk system. A separate magnetic-disk-resident database maps files to locations on the WORM drive, and the design is such that the mapping database can be reconstructed from the WORM-resident permanent data if the magnetic disk ever crashes. The disadvantage of the design is that copies to backup (the WORM system) can take significant time, resulting in snapshots that are far from atomic (e.g., one can still modify the file system while it is being backed up to the WORM drive, which creates the potential for inconsistency).

Tektronix's Optical File Cabinet [Gait 1988, Laskodi et al. 1988] addresses this issue by placing the file system into the WORM drive itself, so there is always an atomic snapshot of the system. The file system uses a logical namespace that is different from the WORM's physical namespace, because the file system transparently maps files to the WORM space, creating a new WORM-resident block and remapping the logical block every time a block is updated by application or Operating System software. Because every block-write allocates a new block in the WORM drive, the files system's logical namespace must necessarily be significantly smaller than the WORM's physically available namespace. It is possible to write a logical block many times, creating many backups of the block on the WORM drive.

In the design described by Quinlan [1991], the best of both worlds is reached by combining the full namespace of the backing store with the magnetic-disk storage medium. The Operating System's file system resides on the backup device, because the file system identifies its blocks using the namespace offered by the backup device. The operating system uses the faster magnetic-disk space as a transparent cache. Instead of creating a new WORM block on every write, the Plan 9 file system creates one on every backup event (e.g., once every night), holding the newly written data on magnetic disk until that point. This is similar in nature to a write-back cache, whereas the Optical File Cabinet is more akin to a write-through cache.

[8] Plan 9 is the (whimsical) name given to version 9 of the Unix operating system.

[9] This research occurred at a time when WORM disks were available, cost roughly the same as magnetic tape, were significantly faster than tape, had a longer shelf-life than magnetic tape, and offered storage densities an order of magnitude larger than rewritable optical media [Hume 1988].

Petitioner Samsung Ex-1009, 0043

1.4.2 Important Issues, Future Directions

Research in cache design is by no means at a stand still, and the future holds numerous avenues for interesting work. For SRAM-based caches in particular, power dissipation and reliability are primary issues. A rule of thumb is that SRAMs typically account for at least one-third of the power dissipated by microprocessors, and the reliability for SRAM is the worst of the technologies surveyed in this book.

On the power-dissipation front, the decline in importance of dynamic power with respect to leakage power changes some traditional rules. For instance, fully associative caches have long been a staple for applications that require high hit rate with a minimum of storage, but they are often avoided due to their high dynamic power dissipation. As leakage power comes to dominate total power, these caches are likely to become more popular, as they can require significantly less storage (and thus fewer transistors) than an equivalently performing direct-mapped or set-associative cache. Fewer transistors means less leakage power.

Taking a different approach, building on-chip caches out of alternative technologies, such as embedded DRAM (eDRAM), is also a promising direction. Though studies in the past have dismissed eDRAM as having unacceptable performance in general-purpose environments, there are several factors at work to counter these initial results, given enough time.

1. The working sets of many high-end applications are growing at a phenomenal pace, making DRAM-based solutions attractive due to their density.
2. The performance of eDRAM technologies is improving to the point that eDRAM matches the performance of off-chip SRAM caches.
3. Embedded DRAM is available in many logic-based processes and even there has a higher reliability than SRAM.
4. Embedded DRAM is expected to dissipate significantly less power, both dynamic and static, compared to SRAM caches.

Another issue for future study is the incredible opportunity for exploring significantly more sophisticated algorithms for content and consistency management as multiple cores and their associated last-level cache controller all move onto the same die. When the last-level cache controller was off-chip (e.g., as part of the system controller), it ran at a significantly slower speed than the microprocessors it served. Moving it onto the same die can increase tenfold the amount of work it can do during the same request. In addition, moving the controller on-chip opens up possibilities for information sharing, giving the controller more access to system state than it ever had before.

Finally, the application of cache principles to new areas that have not traditionally used caches is always a promising area, one that should become very interesting as the sophistication of the various heuristics that can be built grows. For example, the original development of complex block-replacement algorithms, such as least-recently-used (LRU), was done in the context of the operating system's buffer cache, which could afford high-complexity heuristics as the heuristics would be performed in the background of a disk block fetch, a relatively lengthy operation. Such complexity was unthinkable for heuristics in other areas such as solid-state processor caches. Nonetheless, as the technologies from which caches were built in other disciplines grew in capability, these disciplines adopted the same type of analysis and heuristics as the operating system's buffer cache. Currently, the computer-architecture world has explored numerous sophisticated heuristics for prefetching data and dynamic classification of loads. It is only a matter of time before these mechanisms appear elsewhere.

Petitioner Samsung Ex-1009, 0044