

JOHN L. HENNESSY

DAVID A. PATTERSON

COMPUTER ARCHITECTURE

A Quantitative Approach

FIFTH EDITION

MK
MORGAN KAUFMANN

Computer Architecture Formulas

1. $CPU\ time = \text{Instruction count} \times \text{Clock cycles per instruction} \times \text{Clock cycle time}$
2. X is n times faster than Y: $n = \text{Execution time}_Y / \text{Execution time}_X = \text{Performance}_X / \text{Performance}_Y$
3. *Amdahl's Law*: $\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$
4. $\text{Energy}_{\text{dynamic}} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2$
5. $\text{Power}_{\text{dynamic}} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$
6. $\text{Power}_{\text{static}} \propto \text{Current}_{\text{static}} \times \text{Voltage}$
7. $\text{Availability} = \text{Mean Time To Fail} / (\text{Mean Time To Fail} + \text{Mean Time to Repair})$

8. $\text{Die yield} = \text{Wafer yield} \times 1 / (1 + \text{Defects per unit area} \times \text{Die area})^N$
where Wafer yield accounts for wafers that are so bad they need not be tested and N is a parameter called the process-complexity factor, a measure of manufacturing difficulty. N ranges from 11.5 to 15.5 in 2011.

9. *Means—arithmetic (AM), weighted arithmetic (WAM), and geometric (GM):*

$$\text{AM} = \frac{1}{n} \sum_{i=1}^n \text{Time}_i \quad \text{WAM} = \sum_{i=1}^n \text{Weight}_i \times \text{Time}_i \quad \text{GM} = \sqrt[n]{\prod_{i=1}^n \text{Time}_i}$$

where Time_i is the execution time for the i th program of a total of n in the workload, Weight_i is the weighting of the i th program in the workload.

10. $\text{Average memory-access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$
11. $\text{Misses per instruction} = \text{Miss rate} \times \text{Memory access per instruction}$
12. $\text{Cache index size: } 2^{\text{index}} = \text{Cache size} / (\text{Block size} \times \text{Set associativity})$
13. $\text{Power Utilization Effectiveness (PUE) of a Warehouse Scale Computer} = \frac{\text{Total Facility Power}}{\text{IT Equipment Power}}$

Rules of Thumb

1. *Amdahl/Case Rule*: A balanced computer system needs about 1 MB of main memory capacity and 1 megabit per second of I/O bandwidth per MIPS of CPU performance.
2. *90/10 Locality Rule*: A program executes about 90% of its instructions in 10% of its code.
3. *Bandwidth Rule*: Bandwidth grows by at least the square of the improvement in latency.
4. *2:1 Cache Rule*: The miss rate of a direct-mapped cache of size N is about the same as a two-way set-associative cache of size $N/2$.
5. *Dependability Rule*: Design with no single point of failure.
6. *Watt-Year Rule*: The fully burdened cost of a Watt per year in a Warehouse Scale Computer in North America in 2011, including the cost of amortizing the power and cooling infrastructure, is about \$2.

In Praise of *Computer Architecture: A Quantitative Approach* Fifth Edition

“The 5th edition of *Computer Architecture: A Quantitative Approach* continues the legacy, providing students of computer architecture with the most up-to-date information on current computing platforms, and architectural insights to help them design future systems. A highlight of the new edition is the significantly revised chapter on data-level parallelism, which demystifies GPU architectures with clear explanations using traditional computer architecture terminology.”

—Krste Asanović, University of California, Berkeley

“*Computer Architecture: A Quantitative Approach* is a classic that, like fine wine, just keeps getting better. I bought my first copy as I finished up my undergraduate degree and it remains one of my most frequently referenced texts today. When the fourth edition came out, there was so much new material that I needed to get it to stay current in the field. And, as I review the fifth edition, I realize that Hennessy and Patterson have done it again. The entire text is heavily updated and Chapter 6 alone makes this new edition required reading for those wanting to really understand cloud and warehouse scale-computing. Only Hennessy and Patterson have access to the insiders at Google, Amazon, Microsoft, and other cloud computing and internet-scale application providers and there is no better coverage of this important area anywhere in the industry.”

—James Hamilton, Amazon Web Services

“Hennessy and Patterson wrote the first edition of this book when graduate students built computers with 50,000 transistors. Today, warehouse-size computers contain that many servers, each consisting of dozens of independent processors and billions of transistors. The evolution of computer architecture has been rapid and relentless, but *Computer Architecture: A Quantitative Approach* has kept pace, with each edition accurately explaining and analyzing the important emerging ideas that make this field so exciting.”

—James Larus, Microsoft Research

“This new edition adds a superb new chapter on data-level parallelism in vector, SIMD, and GPU architectures. It explains key architecture concepts inside mass-market GPUs, maps them to traditional terms, and compares them with vector and SIMD architectures. It’s timely and relevant with the widespread shift to GPU parallel computing. *Computer Architecture: A Quantitative Approach* furthers its string of firsts in presenting comprehensive architecture coverage of significant new developments!”

—John Nickolls, NVIDIA

“The new edition of this now classic textbook highlights the ascendance of explicit parallelism (data, thread, request) by devoting a whole chapter to each type. The chapter on data parallelism is particularly illuminating: the comparison and contrast between Vector SIMD, instruction level SIMD, and GPU cuts through the jargon associated with each architecture and exposes the similarities and differences between these architectures.”

—Kunle Olukotun, Stanford University

“The fifth edition of *Computer Architecture: A Quantitative Approach* explores the various parallel concepts and their respective tradeoffs. As with the previous editions, this new edition covers the latest technology trends. Two highlighted are the explosive growth of Personal Mobile Devices (PMD) and Warehouse Scale Computing (WSC)—where the focus has shifted towards a more sophisticated balance of performance and energy efficiency as compared with raw performance. These trends are fueling our demand for ever more processing capability which in turn is moving us further down the parallel path.”

—Andrew N. Sloss, Consultant Engineer, ARM
Author of *ARM System Developer's Guide*



Computer Architecture

A Quantitative Approach

Fifth Edition

John L. Hennessy is the tenth president of Stanford University, where he has been a member of the faculty since 1977 in the departments of electrical engineering and computer science. Hennessy is a Fellow of the IEEE and ACM; a member of the National Academy of Engineering, the National Academy of Science, and the American Philosophical Society; and a Fellow of the American Academy of Arts and Sciences. Among his many awards are the 2001 Eckert-Mauchly Award for his contributions to RISC technology, the 2001 Seymour Cray Computer Engineering Award, and the 2000 John von Neumann Award, which he shared with David Patterson. He has also received seven honorary doctorates.

In 1981, he started the MIPS project at Stanford with a handful of graduate students. After completing the project in 1984, he took a leave from the university to cofound MIPS Computer Systems (now MIPS Technologies), which developed one of the first commercial RISC microprocessors. As of 2006, over 2 billion MIPS microprocessors have been shipped in devices ranging from video games and palmtop computers to laser printers and network switches. Hennessy subsequently led the DASH (Director Architecture for Shared Memory) project, which prototyped the first scalable cache coherent multiprocessor; many of the key ideas have been adopted in modern multiprocessors. In addition to his technical activities and university responsibilities, he has continued to work with numerous start-ups both as an early-stage advisor and an investor.

David A. Patterson has been teaching computer architecture at the University of California, Berkeley, since joining the faculty in 1977, where he holds the Pardee Chair of Computer Science. His teaching has been honored by the Distinguished Teaching Award from the University of California, the Karlstrom Award from ACM, and the Mulligan Education Medal and Undergraduate Teaching Award from IEEE. Patterson received the IEEE Technical Achievement Award and the ACM Eckert-Mauchly Award for contributions to RISC, and he shared the IEEE Johnson Information Storage Award for contributions to RAID. He also shared the IEEE John von Neumann Medal and the C & C Prize with John Hennessy. Like his co-author, Patterson is a Fellow of the American Academy of Arts and Sciences, the Computer History Museum, ACM, and IEEE, and he was elected to the National Academy of Engineering, the National Academy of Sciences, and the Silicon Valley Engineering Hall of Fame. He served on the Information Technology Advisory Committee to the U.S. President, as chair of the CS division in the Berkeley EECS department, as chair of the Computing Research Association, and as President of ACM. This record led to Distinguished Service Awards from ACM and CRA.

At Berkeley, Patterson led the design and implementation of RISC I, likely the first VLSI reduced instruction set computer, and the foundation of the commercial SPARC architecture. He was a leader of the Redundant Arrays of Inexpensive Disks (RAID) project, which led to dependable storage systems from many companies. He was also involved in the Network of Workstations (NOW) project, which led to cluster technology used by Internet companies and later to cloud computing. These projects earned three dissertation awards from ACM. His current research projects are Algorithm-Machine-People Laboratory and the Parallel Computing Laboratory, where he is director. The goal of the AMP Lab is develop scalable machine learning algorithms, warehouse-scale-computer-friendly programming models, and crowd-sourcing tools to gain valuable insights quickly from big data in the cloud. The goal of the Par Lab is to develop technologies to deliver scalable, portable, efficient, and productive software for parallel personal mobile devices.

Computer Architecture A Quantitative Approach

Fifth Edition

John L. Hennessy

Stanford University

David A. Patterson

University of California, Berkeley

With Contributions by

Krste Asanović

University of California, Berkeley

Jason D. Bakos

University of South Carolina

Robert P. Colwell

R&E Colwell & Assoc. Inc.

Thomas M. Conte

North Carolina State University

José Duato

Universitat Politècnica de València and Simula

Diana Franklin

University of California, Santa Barbara

David Goldberg

The Scripps Research Institute

Norman P. Jouppi

HP Labs

Sheng Li

HP Labs

Naveen Muralimanohar

HP Labs

Gregory D. Peterson

University of Tennessee

Timothy M. Pinkston

University of Southern California

Parthasarathy Ranganathan

HP Labs

David A. Wood

University of Wisconsin–Madison

Amr Zaky

University of Santa Clara



ELSEVIER

Amsterdam • Boston • Heidelberg • London

New York • Oxford • Paris • San Diego

San Francisco • Singapore • Sydney • Tokyo

MK
MORGAN KAUFMANN

Acquiring Editor: Todd Green
Development Editor: Nate McFadden
Project Manager: Paul Gottehrer
Designer: Joanne Blank

Morgan Kaufmann is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA

© 2012 Elsevier, Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: www.elsevier.com/permissions.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data

Application submitted

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

ISBN: 978-0-12-383872-8

For information on all MK publications
visit our website at www.mkp.com

Printed in the United States of America

Transferred to Digital Print, 2014

Typeset by: diacriTech, Chennai, India

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER

BOOK AID
International

Sabre Foundation

To Andrea, Linda, and our four sons



Foreword

by Luiz André Barroso, Google Inc.

The first edition of Hennessy and Patterson's *Computer Architecture: A Quantitative Approach* was released during my first year in graduate school. I belong, therefore, to that first wave of professionals who learned about our discipline using this book as a compass. Perspective being a fundamental ingredient to a useful Foreword, I find myself at a disadvantage given how much of my own views have been colored by the previous four editions of this book. Another obstacle to clear perspective is that the student-grade reverence for these two superstars of Computer Science has not yet left me, despite (or perhaps because of) having had the chance to get to know them in the years since. These disadvantages are mitigated by my having practiced this trade continuously since this book's first edition, which has given me a chance to enjoy its evolution and enduring relevance.

The last edition arrived just two years after the rampant industrial race for higher CPU clock frequency had come to its official end, with Intel cancelling its 4 GHz single-core developments and embracing multicore CPUs. Two years was plenty of time for John and Dave to present this story not as a random product line update, but as a defining computing technology inflection point of the last decade. That fourth edition had a reduced emphasis on instruction-level parallelism (ILP) in favor of added material on thread-level parallelism, something the current edition takes even further by devoting two chapters to thread- and data-level parallelism while limiting ILP discussion to a single chapter. Readers who are being introduced to new graphics processing engines will benefit especially from the new Chapter 4 which focuses on data parallelism, explaining the different but slowly converging solutions offered by multimedia extensions in general-purpose processors and increasingly programmable graphics processing units. Of notable practical relevance: If you have ever struggled with CUDA terminology check out Figure 4.24 (teaser: "Shared Memory" is really local, while "Global Memory" is closer to what you'd consider shared memory).

Even though we are still in the middle of that multicore technology shift, this edition embraces what appears to be the next major one: cloud computing. In this case, the ubiquity of Internet connectivity and the evolution of compelling Web services are bringing to the spotlight very small devices (smart phones, tablets)

and very large ones (warehouse-scale computing systems). The ARM Cortex A8, a popular CPU for smart phones, appears in Chapter 3's "Putting It All Together" section, and a whole new Chapter 6 is devoted to request- and data-level parallelism in the context of warehouse-scale computing systems. In this new chapter, John and Dave present these new massive clusters as a distinctively new class of computers—an open invitation for computer architects to help shape this emerging field. Readers will appreciate how this area has evolved in the last decade by comparing the Google cluster architecture described in the third edition with the more modern incarnation presented in this version's Chapter 6.

Return customers of this book will appreciate once again the work of two outstanding computer scientists who over their careers have perfected the art of combining an academic's principled treatment of ideas with a deep understanding of leading-edge industrial products and technologies. The authors' success in industrial interactions won't be a surprise to those who have witnessed how Dave conducts his biannual project retreats, forums meticulously crafted to extract the most out of academic-industrial collaborations. Those who recall John's entrepreneurial success with MIPS or bump into him in a Google hallway (as I occasionally do) won't be surprised by it either.

Perhaps most importantly, return and new readers alike will get their money's worth. What has made this book an enduring classic is that each edition is not an update but an extensive revision that presents the most current information and unparalleled insight into this fascinating and quickly changing field. For me, after over twenty years in this profession, it is also another opportunity to experience that student-grade admiration for two remarkable teachers.



Contents

	Foreword	ix
	Preface	xv
	Acknowledgments	xxiii
Chapter 1	Fundamentals of Quantitative Design and Analysis	
	1.1 Introduction	2
	1.2 Classes of Computers	5
	1.3 Defining Computer Architecture	11
	1.4 Trends in Technology	17
	1.5 Trends in Power and Energy in Integrated Circuits	21
	1.6 Trends in Cost	27
	1.7 Dependability	33
	1.8 Measuring, Reporting, and Summarizing Performance	36
	1.9 Quantitative Principles of Computer Design	44
	1.10 Putting It All Together: Performance, Price, and Power	52
	1.11 Fallacies and Pitfalls	55
	1.12 Concluding Remarks	59
	1.13 Historical Perspectives and References	61
	Case Studies and Exercises by Diana Franklin	61
Chapter 2	Memory Hierarchy Design	
	2.1 Introduction	72
	2.2 Ten Advanced Optimizations of Cache Performance	78
	2.3 Memory Technology and Optimizations	96
	2.4 Protection: Virtual Memory and Virtual Machines	105
	2.5 Crosscutting Issues: The Design of Memory Hierarchies	112
	2.6 Putting It All Together: Memory Hierarchies in the ARM Cortex-A8 and Intel Core i7	113
	2.7 Fallacies and Pitfalls	125

2.8	Concluding Remarks: Looking Ahead	129
2.9	Historical Perspective and References	131
	Case Studies and Exercises by Norman P. Jouppi, Naveen Muralimanohar, and Sheng Li	131
Chapter 3	Instruction-Level Parallelism and Its Exploitation	
3.1	Instruction-Level Parallelism: Concepts and Challenges	148
3.2	Basic Compiler Techniques for Exposing ILP	156
3.3	Reducing Branch Costs with Advanced Branch Prediction	162
3.4	Overcoming Data Hazards with Dynamic Scheduling	167
3.5	Dynamic Scheduling: Examples and the Algorithm	176
3.6	Hardware-Based Speculation	183
3.7	Exploiting ILP Using Multiple Issue and Static Scheduling	192
3.8	Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation	197
3.9	Advanced Techniques for Instruction Delivery and Speculation	202
3.10	Studies of the Limitations of ILP	213
3.11	Cross-Cutting Issues: ILP Approaches and the Memory System	221
3.12	Multithreading: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput	223
3.13	Putting It All Together: The Intel Core i7 and ARM Cortex-A8	233
3.14	Fallacies and Pitfalls	241
3.15	Concluding Remarks: What's Ahead?	245
3.16	Historical Perspective and References	247
	Case Studies and Exercises by Jason D. Bakos and Robert P. Colwell	247
Chapter 4	Data-Level Parallelism in Vector, SIMD, and GPU Architectures	
4.1	Introduction	262
4.2	Vector Architecture	264
4.3	SIMD Instruction Set Extensions for Multimedia	282
4.4	Graphics Processing Units	288
4.5	Detecting and Enhancing Loop-Level Parallelism	315
4.6	Crosscutting Issues	322
4.7	Putting It All Together: Mobile versus Server GPUs and Tesla versus Core i7	323
4.8	Fallacies and Pitfalls	330
4.9	Concluding Remarks	332
4.10	Historical Perspective and References	334
	Case Study and Exercises by Jason D. Bakos	334
Chapter 5	Thread-Level Parallelism	
5.1	Introduction	344
5.2	Centralized Shared-Memory Architectures	351
5.3	Performance of Symmetric Shared-Memory Multiprocessors	366

5.4	Distributed Shared-Memory and Directory-Based Coherence	378
5.5	Synchronization: The Basics	386
5.6	Models of Memory Consistency: An Introduction	392
5.7	Crosscutting Issues	395
5.8	Putting It All Together: Multicore Processors and Their Performance	400
5.9	Fallacies and Pitfalls	405
5.10	Concluding Remarks	409
5.11	Historical Perspectives and References	412
	Case Studies and Exercises by Amr Zaky and David A. Wood	412

Chapter 6 **Warehouse-Scale Computers to Exploit Request-Level and Data-Level Parallelism**

6.1	Introduction	432
6.2	Programming Models and Workloads for Warehouse-Scale Computers	436
6.3	Computer Architecture of Warehouse-Scale Computers	441
6.4	Physical Infrastructure and Costs of Warehouse-Scale Computers	446
6.5	Cloud Computing: The Return of Utility Computing	455
6.6	Crosscutting Issues	461
6.7	Putting It All Together: A Google Warehouse-Scale Computer	464
6.8	Fallacies and Pitfalls	471
6.9	Concluding Remarks	475
6.10	Historical Perspectives and References	476
	Case Studies and Exercises by Parthasarathy Ranganathan	476

Appendix A **Instruction Set Principles**

A.1	Introduction	A-2
A.2	Classifying Instruction Set Architectures	A-3
A.3	Memory Addressing	A-7
A.4	Type and Size of Operands	A-13
A.5	Operations in the Instruction Set	A-14
A.6	Instructions for Control Flow	A-16
A.7	Encoding an Instruction Set	A-21
A.8	Crosscutting Issues: The Role of Compilers	A-24
A.9	Putting It All Together: The MIPS Architecture	A-32
A.10	Fallacies and Pitfalls	A-39
A.11	Concluding Remarks	A-45
A.12	Historical Perspective and References	A-47
	Exercises by Gregory D. Peterson	A-47

Appendix B **Review of Memory Hierarchy**

B.1	Introduction	B-2
B.2	Cache Performance	B-16
B.3	Six Basic Cache Optimizations	B-22

	B.4	Virtual Memory	B-40
	B.5	Protection and Examples of Virtual Memory	B-49
	B.6	Fallacies and Pitfalls	B-57
	B.7	Concluding Remarks	B-59
	B.8	Historical Perspective and References	B-59
		Exercises by Amr Zaky	B-60
Appendix C	Pipelining: Basic and Intermediate Concepts		
	C.1	Introduction	C-2
	C.2	The Major Hurdle of Pipelining—Pipeline Hazards	C-11
	C.3	How Is Pipelining Implemented?	C-30
	C.4	What Makes Pipelining Hard to Implement?	C-43
	C.5	Extending the MIPS Pipeline to Handle Multicycle Operations	C-51
	C.6	Putting It All Together: The MIPS R4000 Pipeline	C-61
	C.7	Crosscutting Issues	C-70
	C.8	Fallacies and Pitfalls	C-80
	C.9	Concluding Remarks	C-81
	C.10	Historical Perspective and References	C-81
		Updated Exercises by Diana Franklin	C-82
	Online Appendices		
Appendix D	Storage Systems		
Appendix E	Embedded Systems		
	<i>By Thomas M. Conte</i>		
Appendix F	Interconnection Networks		
	<i>Revised by Timothy M. Pinkston and José Duato</i>		
Appendix G	Vector Processors in More Depth		
	<i>Revised by Krste Asanovic</i>		
Appendix H	Hardware and Software for VLIW and EPIC		
Appendix I	Large-Scale Multiprocessors and Scientific Applications		
Appendix J	Computer Arithmetic		
	<i>by David Goldberg</i>		
Appendix K	Survey of Instruction Set Architectures		
Appendix L	Historical Perspectives and References		
	References		R-1
	Index		I-1

2

Memory Hierarchy Design

Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available. ... We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

A. W. Burks, H. H. Goldstine,
and J. von Neumann

*Preliminary Discussion of the
Logical Design of an Electronic
Computing Instrument (1946)*

2.1 Introduction

Computer pioneers correctly predicted that programmers would want unlimited amounts of fast memory. An economical solution to that desire is a *memory hierarchy*, which takes advantage of locality and trade-offs in the cost-performance of memory technologies. The *principle of locality*, presented in the first chapter, says that most programs do not access all code or data uniformly. Locality occurs in time (*temporal locality*) and in space (*spatial locality*). This principle, plus the guideline that for a given implementation technology and power budget smaller hardware can be made faster, led to hierarchies based on memories of different speeds and sizes. Figure 2.1 shows a multilevel memory hierarchy, including typical sizes and speeds of access.

Since fast memory is expensive, a memory hierarchy is organized into several levels—each smaller, faster, and more expensive per byte than the next lower level, which is farther from the processor. The goal is to provide a memory system with cost per byte almost as low as the cheapest level of memory and speed almost as fast as the fastest level. In most cases (but not all), the data contained in a lower level are a superset of the next higher level. This property, called the *inclusion property*, is always required for the lowest level of the hierarchy, which consists of main memory in the case of caches and disk memory in the case of virtual memory.

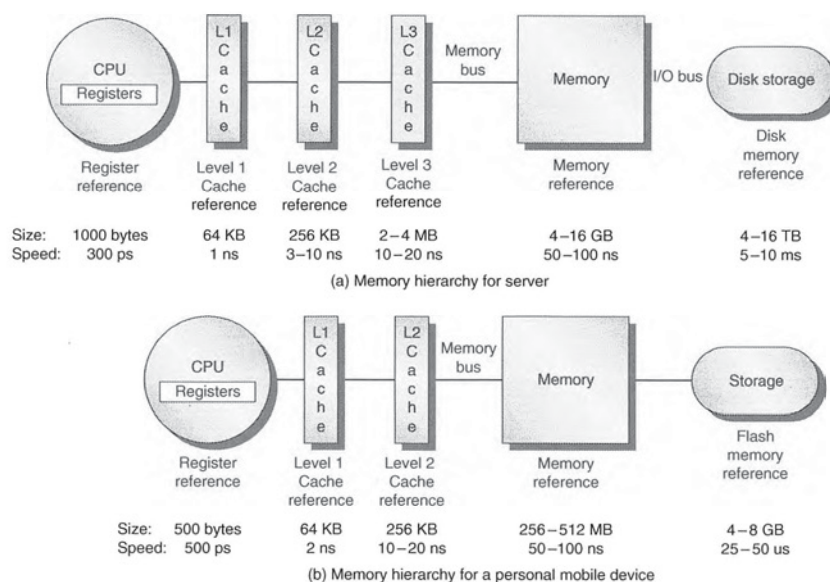


Figure 2.1 The levels in a typical memory hierarchy in a server computer shown on top (a) and in a personal mobile device (PMD) on the bottom (b). As we move farther away from the processor, the memory in the level below becomes slower and larger. Note that the time units change by a factor of 10^9 —from picoseconds to milliseconds—and that the size units change by a factor of 10^{12} —from bytes to terabytes. The PMD has a slower clock rate and smaller caches and main memory. A key difference is that servers and desktops use disk storage as the lowest level in the hierarchy while PMDs use Flash, which is built from EEPROM technology.

The importance of the memory hierarchy has increased with advances in performance of processors. Figure 2.2 plots single processor performance projections against the historical performance improvement in time to access main memory. The processor line shows the increase in memory requests per second on average (i.e., the inverse of the latency between memory references), while the memory line shows the increase in DRAM accesses per second (i.e., the inverse of the DRAM access latency). The situation in a uniprocessor is actually somewhat worse, since the peak memory access rate is faster than the average rate, which is what is plotted.

More recently, high-end processors have moved to multiple cores, further increasing the bandwidth requirements versus single cores. In fact, the aggregate peak bandwidth essentially grows as the numbers of cores grows. A modern high-end processor such as the Intel Core i7 can generate two data memory references per core each clock cycle; with four cores and a 3.2 GHz clock rate, the i7 can generate a peak of 25.6 billion 64-bit data memory references per second, in addition to a peak instruction demand of about 12.8 billion 128-bit instruction references; this is a total peak bandwidth of 409.6 GB/sec! This incredible bandwidth is achieved by multiporting and pipelining the caches; by the use of multiple levels of caches, using separate first- and sometimes second-level caches per core; and by using a separate instruction and data cache at the first level. In contrast, the peak bandwidth to DRAM main memory is only 6% of this (25 GB/sec).

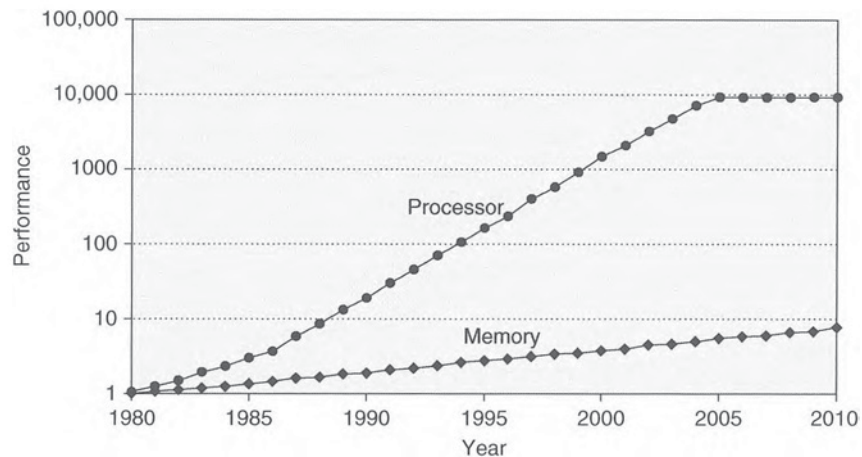


Figure 2.2 Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time. Note that the vertical axis must be on a logarithmic scale to record the size of the processor–DRAM performance gap. The memory baseline is 64 KB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 2.13 on page 99). The processor line assumes a 1.25 improvement per year until 1986, a 1.52 improvement until 2000, a 1.20 improvement between 2000 and 2005, and no change in processor performance (on a per-core basis) between 2005 and 2010; see Figure 1.1 in Chapter 1.

Traditionally, designers of memory hierarchies focused on optimizing average memory access time, which is determined by the cache access time, miss rate, and miss penalty. More recently, however, power has become a major consideration. In high-end microprocessors, there may be 10 MB or more of on-chip cache, and a large second- or third-level cache will consume significant power both as leakage when not operating (called *static power*) and as active power, as when performing a read or write (called *dynamic power*), as described in Section 2.3. The problem is even more acute in processors in PMDs where the CPU is less aggressive and the power budget may be 20 to 50 times smaller. In such cases, the caches can account for 25% to 50% of the total power consumption. Thus, more designs must consider both performance and power trade-offs, and we will examine both in this chapter.

Basics of Memory Hierarchies: A Quick Review

The increasing size and thus importance of this gap led to the migration of the basics of memory hierarchy into undergraduate courses in computer architecture, and even to courses in operating systems and compilers. Thus, we'll start with a quick review of caches and their operation. The bulk of the chapter, however, describes more advanced innovations that attack the processor-memory performance gap.

When a word is not found in the cache, the word must be fetched from a lower level in the hierarchy (which may be another cache or the main memory) and placed in the cache before continuing. Multiple words, called a *block* (or *line*), are moved for efficiency reasons, and because they are likely to be needed soon due to spatial locality. Each cache block includes a *tag* to indicate which memory address it corresponds to.

A key design decision is where blocks (or lines) can be placed in a cache. The most popular scheme is *set associative*, where a *set* is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set. Finding a block consists of first mapping the block address to the set and then searching the set—usually in parallel—to find the block. The set is chosen by the address of the data:

$$(\text{Block address}) \bmod (\text{Number of sets in cache})$$

If there are n blocks in a set, the cache placement is called *n-way set associative*. The end points of set associativity have their own names. A *direct-mapped* cache has just one block per set (so a block is always placed in the same location), and a *fully associative* cache has just one set (so a block can be placed anywhere).

Caching data that is only read is easy, since the copy in the cache and memory will be identical. Caching writes is more difficult; for example, how can the copy in the cache and memory be kept consistent? There are two main strategies. A *write-through* cache updates the item in the cache *and* writes through to update

main memory. A *write-back* cache only updates the copy in the cache. When the block is about to be replaced, it is copied back to memory. Both write strategies can use a *write buffer* to allow the cache to proceed as soon as the data are placed in the buffer rather than wait the full latency to write the data into memory.

One measure of the benefits of different cache organizations is miss rate. *Miss rate* is simply the fraction of cache accesses that result in a miss—that is, the number of accesses that miss divided by the number of accesses.

To gain insights into the causes of high miss rates, which can inspire better cache designs, the three Cs model sorts all misses into three simple categories:

- *Compulsory*—The very first access to a block *cannot* be in the cache, so the block must be brought into the cache. Compulsory misses are those that occur even if you had an infinite sized cache.
- *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.
- *Conflict*—If the block placement strategy is not fully associative, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if multiple blocks map to its set and accesses to the different blocks are intermingled.

Figures B.8 and B.9 on pages B-24 and B-25 show the relative frequency of cache misses broken down by the three Cs. As we will see in Chapters 3 and 5, multithreading and multiple cores add complications for caches, both increasing the potential for capacity misses as well as adding a fourth C, for *coherency* misses due to cache flushes to keep multiple caches coherent in a multiprocessor; we will consider these issues in Chapter 5.

Alas, miss rate can be a misleading measure for several reasons. Hence, some designers prefer measuring *misses per instruction* rather than misses per memory reference (miss rate). These two are related:

$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Miss rate} \times \text{Memory accesses}}{\text{Instruction count}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

(It is often reported as misses per 1000 instructions to use integers instead of fractions.)

The problem with both measures is that they don't factor in the cost of a miss. A better measure is the *average memory access time*:

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

where *hit time* is the time to hit in the cache and *miss penalty* is the time to replace the block from memory (that is, the cost of a miss). Average memory access time is still an indirect measure of performance; although it is a better measure than miss rate, it is not a substitute for execution time. In Chapter 3 we will see that speculative processors may execute other instructions during a miss, thereby reducing the

effective miss penalty. The use of multithreading (introduced in Chapter 3) also allows a processor to tolerate misses without being forced to idle. As we will examine shortly, to take advantage of such latency tolerating techniques we need caches that can service requests while handling an outstanding miss.

If this material is new to you, or if this quick review moves too quickly, see Appendix B. It covers the same introductory material in more depth and includes examples of caches from real computers and quantitative evaluations of their effectiveness.

Section B.3 in Appendix B presents six basic cache optimizations, which we quickly review here. The appendix also gives quantitative examples of the benefits of these optimizations. We also comment briefly on the power implications of these trade-offs.

1. *Larger block size to reduce miss rate*—The simplest way to reduce the miss rate is to take advantage of spatial locality and increase the block size. Larger blocks reduce compulsory misses, but they also increase the miss penalty. Because larger blocks lower the number of tags, they can slightly reduce static power. Larger block sizes can also increase capacity or conflict misses, especially in smaller caches. Choosing the right block size is a complex trade-off that depends on the size of cache and the miss penalty.
2. *Bigger caches to reduce miss rate*—The obvious way to reduce capacity misses is to increase cache capacity. Drawbacks include potentially longer hit time of the larger cache memory and higher cost and power. Larger caches increase both static and dynamic power.
3. *Higher associativity to reduce miss rate*—Obviously, increasing associativity reduces conflict misses. Greater associativity can come at the cost of increased hit time. As we will see shortly, associativity also increases power consumption.
4. *Multilevel caches to reduce miss penalty*—A difficult decision is whether to make the cache hit time fast, to keep pace with the high clock rate of processors, or to make the cache large to reduce the gap between the processor accesses and main memory accesses. Adding another level of cache between the original cache and memory simplifies the decision (see Figure 2.3). The first-level cache can be small enough to match a fast clock cycle time, yet the second-level (or third-level) cache can be large enough to capture many accesses that would go to main memory. The focus on misses in second-level caches leads to larger blocks, bigger capacity, and higher associativity. Multilevel caches are more power efficient than a single aggregate cache. If L1 and L2 refer, respectively, to first- and second-level caches, we can redefine the average memory access time:

$$\text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

5. *Giving priority to read misses over writes to reduce miss penalty*—A write buffer is a good place to implement this optimization. Write buffers create hazards because they hold the updated value of a location needed on a read

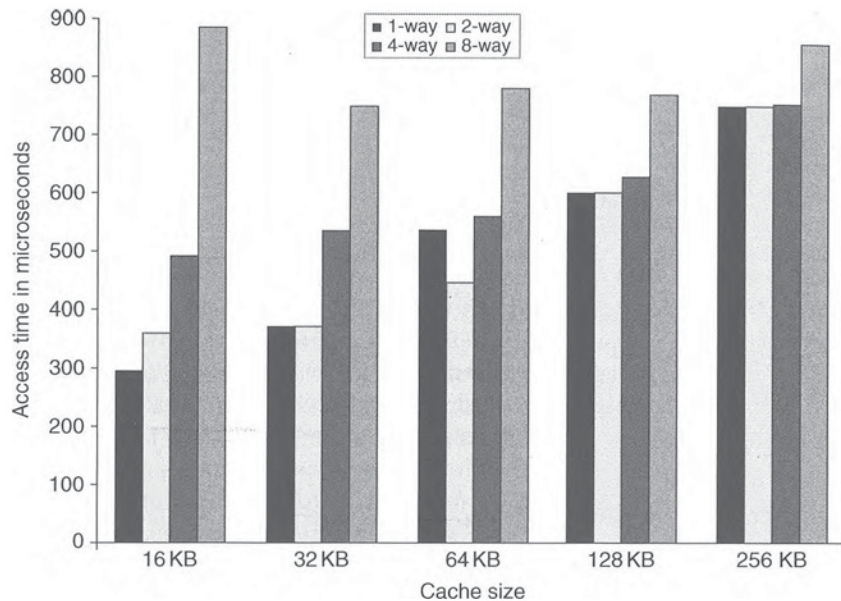


Figure 2.3 Access times generally increase as cache size and associativity are increased. These data come from the CACTI model 6.5 by Tarjan, Thoziyoor, and Jouppi [2005]. The data assume a 40 nm feature size (which is between the technology used in Intel’s fastest and second fastest versions of the i7 and the same as the technology used in the fastest ARM embedded processors), a single bank, and 64-byte blocks. The assumptions about cache layout and the complex trade-offs between interconnect delays (that depend on the size of a cache block being accessed) and the cost of tag checks and multiplexing lead to results that are occasionally surprising, such as the lower access time for a 64 KB with two-way set associativity versus direct mapping. Similarly, the results with eight-way set associativity generate unusual behavior as cache size is increased. Since such observations are highly dependent on technology and detailed design assumptions, tools such as CACTI serve to reduce the search space rather than precision analysis of the trade-offs.

miss—that is, a read-after-write hazard through memory. One solution is to check the contents of the write buffer on a read miss. If there are no conflicts, and if the memory system is available, sending the read before the writes reduces the miss penalty. Most processors give reads priority over writes. This choice has little effect on power consumption.

6. *Avoiding address translation during indexing of the cache to reduce hit time*—Caches must cope with the translation of a virtual address from the processor to a physical address to access memory. (Virtual memory is covered in Sections 2.4 and B.4.) A common optimization is to use the page offset—the part that is identical in both virtual and physical addresses—to index the cache, as described in Appendix B, page B-38. This virtual index/physical tag method introduces some system complications and/or

limitations on the size and structure of the L1 cache, but the advantages of removing the translation lookaside buffer (TLB) access from the critical path outweigh the disadvantages.

Note that each of the six optimizations above has a potential disadvantage that can lead to increased, rather than decreased, average memory access time.

The rest of this chapter assumes familiarity with the material above and the details in Appendix B. In the Putting It All Together section, we examine the memory hierarchy for a microprocessor designed for a high-end server, the Intel Core i7, as well as one designed for use in a PMD, the Arm Cortex-A8, which is the basis for the processor used in the Apple iPad and several high-end smartphones. Within each of these classes, there is a significant diversity in approach due to the intended use of the computer. While the high-end processor used in the server has more cores and bigger caches than the Intel processors designed for desktop uses, the processors have similar architectures. The differences are driven by performance and the nature of the workload; desktop computers are primarily running one application at a time on top of an operating system for a single user, whereas server computers may have hundreds of users running potentially dozens of applications simultaneously. Because of these workload differences, desktop computers are generally concerned more with average latency from the memory hierarchy, whereas server computers are also concerned about memory bandwidth. Even within the class of desktop computers there is wide diversity from lower end netbooks with scaled-down processors more similar to those found in high-end PMDs, to high-end desktops whose processors contain multiple cores and whose organization resembles that of a low-end server.

In contrast, PMDs not only serve one user but generally also have smaller operating systems, usually less multitasking (running of several applications simultaneously), and simpler applications. PMDs also typically use Flash memory rather than disks, and most consider both performance and energy consumption, which determines battery life.

2.2

Ten Advanced Optimizations of Cache Performance

The average memory access time formula above gives us three metrics for cache optimizations: hit time, miss rate, and miss penalty. Given the recent trends, we add cache bandwidth and power consumption to this list. We can classify the ten advanced cache optimizations we examine into five categories based on these metrics:

1. *Reducing the hit time*—Small and simple first-level caches and way-prediction. Both techniques also generally decrease power consumption.
2. *Increasing cache bandwidth*—Pipelined caches, multibanked caches, and nonblocking caches. These techniques have varying impacts on power consumption.

3. *Reducing the miss penalty*—Critical word first and merging write buffers. These optimizations have little impact on power.
4. *Reducing the miss rate*—Compiler optimizations. Obviously any improvement at compile time improves power consumption.
5. *Reducing the miss penalty or miss rate via parallelism*—Hardware prefetching and compiler prefetching. These optimizations generally increase power consumption, primarily due to prefetched data that are unused.

In general, the hardware complexity increases as we go through these optimizations. In addition, several of the optimizations require sophisticated compiler technology. We will conclude with a summary of the implementation complexity and the performance benefits of the ten techniques presented in Figure 2.11 on page 96. Since some of these are straightforward, we cover them briefly; others require more description.

First Optimization: Small and Simple First-Level Caches to Reduce Hit Time and Power

The pressure of both a fast clock cycle and power limitations encourages limited size for first-level caches. Similarly, use of lower levels of associativity can reduce both hit time and power, although such trade-offs are more complex than those involving size.

The critical timing path in a cache hit is the three-step process of addressing the tag memory using the index portion of the address, comparing the read tag value to the address, and setting the multiplexor to choose the correct data item if the cache is set associative. Direct-mapped caches can overlap the tag check with the transmission of the data, effectively reducing hit time. Furthermore, lower levels of associativity will usually reduce power because fewer cache lines must be accessed.

Although the total amount of on-chip cache has increased dramatically with new generations of microprocessors, due to the clock rate impact arising from a larger L1 cache, the size of the L1 caches has recently increased either slightly or not at all. In many recent processors, designers have opted for more associativity rather than larger caches. An additional consideration in choosing the associativity is the possibility of eliminating address aliases; we discuss this shortly.

One approach to determining the impact on hit time and power consumption in advance of building a chip is to use CAD tools. CACTI is a program to estimate the access time and energy consumption of alternative cache structures on CMOS microprocessors within 10% of more detailed CAD tools. For a given minimum feature size, CACTI estimates the hit time of caches as cache size varies, associativity, number of read/write ports, and more complex parameters. Figure 2.3 shows the estimated impact on hit time as cache size and associativity are varied. Depending on cache size, for these parameters the model suggests that the hit time for direct mapped is slightly faster than two-way set associative and

that two-way set associative is 1.2 times faster than four-way and four-way is 1.4 times faster than eight-way. Of course, these estimates depend on technology as well as the size of the cache.

Example Using the data in Figure B.8 in Appendix B and Figure 2.3, determine whether a 32 KB four-way set associative L1 cache has a faster memory access time than a 32 KB two-way set associative L1 cache. Assume the miss penalty to L2 is 15 times the access time for the faster L1 cache. Ignore misses beyond L2. Which has the faster average memory access time?

Answer Let the access time for the two-way set associative cache be 1. Then, for the two-way cache:

$$\begin{aligned}\text{Average memory access time}_{2\text{-way}} &= \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \\ &= 1 + 0.038 \times 15 = 1.38\end{aligned}$$

For the four-way cache, the access time is 1.4 times longer. The elapsed time of the miss penalty is $15/1.4 = 10.1$. Assume 10 for simplicity:

$$\begin{aligned}\text{Average memory access time}_{4\text{-way}} &= \text{Hit time}_{2\text{-way}} \times 1.4 + \text{Miss rate} \times \text{Miss penalty} \\ &= 1.4 + 0.037 \times 10 = 1.77\end{aligned}$$

Clearly, the higher associativity looks like a bad trade-off; however, since cache access in modern processors is often pipelined, the exact impact on the clock cycle time is difficult to assess.

Energy consumption is also a consideration in choosing both the cache size and associativity, as Figure 2.4 shows. The energy cost of higher associativity ranges from more than a factor of 2 to negligible in caches of 128 KB or 256 KB when going from direct mapped to two-way set associative.

In recent designs, there are three other factors that have led to the use of higher associativity in first-level caches. First, many processors take at least two clock cycles to access the cache and thus the impact of a longer hit time may not be critical. Second, to keep the TLB out of the critical path (a delay that would be larger than that associated with increased associativity), almost all L1 caches should be virtually indexed. This limits the size of the cache to the page size times the associativity, because then only the bits within the page are used for the index. There are other solutions to the problem of indexing the cache before address translation is completed, but increasing the associativity, which also has other benefits, is the most attractive. Third, with the introduction of multithreading (see Chapter 3), conflict misses can increase, making higher associativity more attractive.

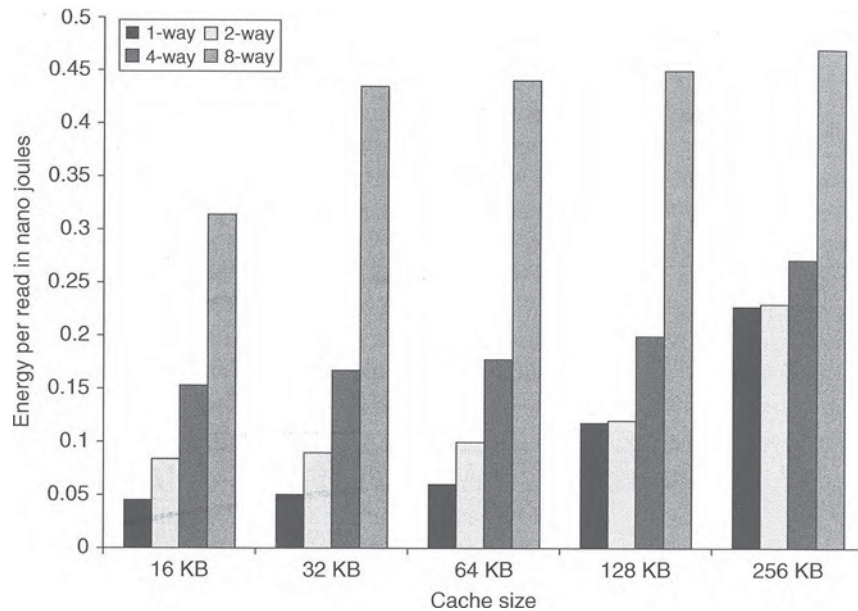


Figure 2.4 Energy consumption per read increases as cache size and associativity are increased. As in the previous figure, CACTI is used for the modeling with the same technology parameters. The large penalty for eight-way set associative caches is due to the cost of reading out eight tags and the corresponding data in parallel.

Second Optimization: Way Prediction to Reduce Hit Time

Another approach reduces conflict misses and yet maintains the hit speed of direct-mapped cache. In *way prediction*, extra bits are kept in the cache to predict the way, or block within the set of the *next* cache access. This prediction means the multiplexor is set early to select the desired block, and only a single tag comparison is performed that clock cycle in-parallel with reading the cache data. A miss results in checking the other blocks for matches in the next clock cycle.

Added to each block of a cache are block predictor bits. The bits select which of the blocks to try on the *next* cache access. If the predictor is correct, the cache access latency is the fast hit time. If not, it tries the other block, changes the way predictor, and has a latency of one extra clock cycle. Simulations suggest that set prediction accuracy is in excess of 90% for a two-way set associative cache and 80% for a four-way set associative cache, with better accuracy on I-caches than D-caches. Way prediction yields lower average memory access time for a two-way set associative cache if it is at least 10% faster, which is quite likely. Way prediction was first used in the MIPS R10000 in the mid-1990s. It is popular in processors that use two-way set associativity and is used in the ARM Cortex-A8 with four-way set associative caches. For very fast processors, it may be challenging to implement the one cycle stall that is critical to keeping the way prediction penalty small.

An extended form of way prediction can also be used to reduce power consumption by using the way prediction bits to decide which cache block to actually access (the way prediction bits are essentially extra address bits); this approach, which might be called *way selection*, saves power when the way prediction is correct but adds significant time on a way misprediction, since the access, not just the tag match and selection, must be repeated. Such an optimization is likely to make sense only in low-power processors. Inoue, Ishihara, and Murakami [1999] estimated that using the way selection approach with a four-way set associative cache increases the average access time for the I-cache by 1.04 and for the D-cache by 1.13 on the SPEC95 benchmarks, but it yields an average cache power consumption relative to a normal four-way set associative cache that is 0.28 for the I-cache and 0.35 for the D-cache. One significant drawback for way selection is that it makes it difficult to pipeline the cache access.

Example Assume that there are half as many D-cache accesses as I-cache accesses, and that the I-cache and D-cache are responsible for 25% and 15% of the processor's power consumption in a normal four-way set associative implementation. Determine if way selection improves performance per watt based on the estimates from the study above.

Answer For the I-cache, the savings in power is $25 \times 0.28 = 0.07$ of the total power, while for the D-cache it is $15 \times 0.35 = 0.05$ for a total savings of 0.12. The way prediction version requires 0.88 of the power requirement of the standard 4-way cache. The increase in cache access time is the increase in I-cache average access time plus one-half the increase in D-cache access time, or $1.04 + 0.5 \times 0.13 = 1.11$ times longer. This result means that way selection has 0.90 of the performance of a standard four-way cache. Thus, way selection improves performance per joule very slightly by a ratio of $0.90/0.88 = 1.02$. This optimization is best used where power rather than performance is the key objective.

Third Optimization: Pipelined Cache Access to Increase Cache Bandwidth

This optimization is simply to pipeline cache access so that the effective latency of a first-level cache hit can be multiple clock cycles, giving fast clock cycle time and high bandwidth but slow hits. For example, the pipeline for the instruction cache access for Intel Pentium processors in the mid-1990s took 1 clock cycle, for the Pentium Pro through Pentium III in the mid-1990s through 2000 it took 2 clocks, and for the Pentium 4, which became available in 2000, and the current Intel Core i7 it takes 4 clocks. This change increases the number of pipeline stages, leading to a greater penalty on mispredicted branches and more clock cycles between issuing the load and using the data (see Chapter 3), but it does make it easier to incorporate high degrees of associativity.

Fourth Optimization: Nonblocking Caches to Increase Cache Bandwidth

For pipelined computers that allow out-of-order execution (discussed in Chapter 3), the processor need not stall on a data cache miss. For example, the processor could continue fetching instructions from the instruction cache while waiting for the data cache to return the missing data. A *nonblocking cache* or *lookup-free cache* escalates the potential benefits of such a scheme by allowing the data cache to continue to supply cache hits during a miss. This “hit under miss” optimization reduces the effective miss penalty by being helpful during a miss instead of ignoring the requests of the processor. A subtle and complex option is that the cache may further lower the effective miss penalty if it can overlap multiple misses: a “hit under multiple miss” or “miss under miss” optimization. The second option is beneficial only if the memory system can service multiple misses; most high-performance processors (such as the Intel Core i7) usually support both, while lower end processors, such as the ARM A8, provide only limited nonblocking support in L2.

To examine the effectiveness of nonblocking caches in reducing the cache miss penalty, Farkas and Jouppi [1994] did a study assuming 8 KB caches with a 14-cycle miss penalty; they observed a reduction in the effective miss penalty of 20% for the SPECINT92 benchmarks and 30% for the SPECFP92 benchmarks when allowing one hit under miss.

Li, Chen, Brockman, and Jouppi [2011] recently updated this study to use a multilevel cache, more modern assumptions about miss penalties, and the larger and more demanding SPEC2006 benchmarks. The study was done assuming a model based on a single core of an Intel i7 (see Section 2.6) running the SPEC2006 benchmarks. Figure 2.5 shows the reduction in data cache access latency when allowing 1, 2, and 64 hits under a miss; the caption describes further details of the memory system. The larger caches and the addition of an L3 cache since the earlier study have reduced the benefits with the SPECINT2006 benchmarks showing an average reduction in cache latency of about 9% and the SPECFP2006 benchmarks about 12.5%.

Example Which is more important for floating-point programs: two-way set associativity or hit under one miss for the primary data caches? What about integer programs? Assume the following average miss rates for 32 KB data caches: 5.2% for floating-point programs with a direct-mapped cache, 4.9% for these programs with a two-way set associative cache, 3.5% for integer programs with a direct-mapped cache, and 3.2% for integer programs with a two-way set associative cache. Assume the miss penalty to L2 is 10 cycles, and the L2 misses and penalties are the same.

Answer For floating-point programs, the average memory stall times are

$$\text{Miss rate}_{\text{DM}} \times \text{Miss penalty} = 5.2\% \times 10 = 0.52$$

$$\text{Miss rate}_{2\text{-way}} \times \text{Miss penalty} = 4.9\% \times 10 = 0.49$$

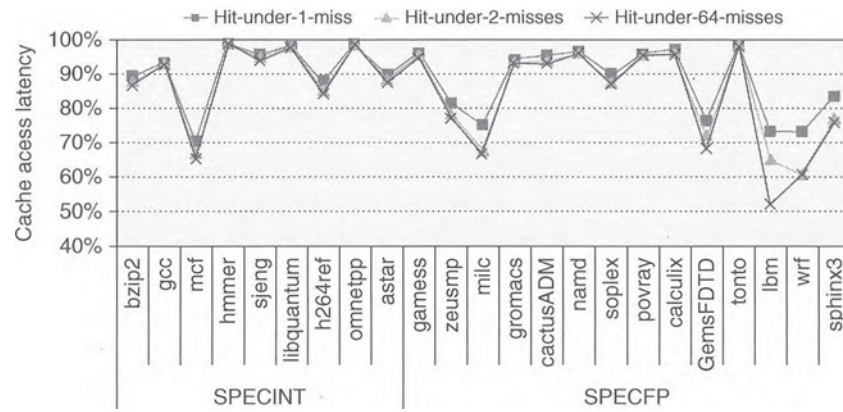


Figure 2.5 The effectiveness of a nonblocking cache is evaluated by allowing 1, 2, or 64 hits under a cache miss with 9 SPECINT (on the left) and 9 SPEC FP (on the right) benchmarks. The data memory system modeled after the Intel i7 consists of a 32KB L1 cache with a four cycle access latency. The L2 cache (shared with instructions) is 256 KB with a 10 clock cycle access latency. The L3 is 2 MB and a 36-cycle access latency. All the caches are eight-way set associative and have a 64-byte block size. Allowing one hit under miss reduces the miss penalty by 9% for the integer benchmarks and 12.5% for the floating point. Allowing a second hit improves these results to 10% and 16%, and allowing 64 results in little additional improvement.

The cache access latency (including stalls) for two-way associativity is 0.49/0.52 or 94% of direct-mapped cache. The caption of Figure 2.5 says hit under one miss reduces the average data cache access latency for floating point programs to 87.5% of a blocking cache. Hence, for floating-point programs, the direct mapped data cache supporting one hit under one miss gives better performance than a two-way set-associative cache that blocks on a miss.

For integer programs, the calculation is

$$\text{Miss rate}_{\text{DM}} \times \text{Miss penalty} = 3.5\% \times 10 = 0.35$$

$$\text{Miss rate}_{2\text{-way}} \times \text{Miss penalty} = 3.2\% \times 10 = 0.32$$

The data cache access latency of a two-way set associative cache is thus 0.32/0.35 or 91% of direct-mapped cache, while the reduction in access latency when allowing a hit under one miss is 9%, making the two choices about equal.

The real difficulty with performance evaluation of nonblocking caches is that a cache miss does not necessarily stall the processor. In this case, it is difficult to judge the impact of any single miss and hence to calculate the average memory access time. The effective miss penalty is not the sum of the misses but the non-overlapped time that the processor is stalled. The benefit of nonblocking caches is complex, as it depends upon the miss penalty when there are multiple misses, the memory reference pattern, and how many instructions the processor can execute with a miss outstanding.

In general, out-of-order processors are capable of hiding much of the miss penalty of an L1 data cache miss that hits in the L2 cache but are not capable of hiding a significant fraction of a lower level cache miss. Deciding how many outstanding misses to support depends on a variety of factors:

- The temporal and spatial locality in the miss stream, which determines whether a miss can initiate a new access to a lower level cache or to memory
- The bandwidth of the responding memory or cache
- To allow more outstanding misses at the lowest level of the cache (where the miss time is the longest) requires supporting at least that many misses at a higher level, since the miss must initiate at the highest level cache
- The latency of the memory system

The following simplified example shows the key idea.

Example Assume a main memory access time of 36 ns and a memory system capable of a sustained transfer rate of 16 GB/sec. If the block size is 64 bytes, what is the maximum number of outstanding misses we need to support assuming that we can maintain the peak bandwidth given the request stream and that accesses never conflict. If the probability of a reference colliding with one of the previous four is 50%, and we assume that the access has to wait until the earlier access completes, estimate the number of maximum outstanding references. For simplicity, ignore the time between misses.

Answer In the first case, assuming that we can maintain the peak bandwidth, the memory system can support $(16 \times 10^9)/64 = 250$ million references per second. Since each reference takes 36 ns, we can support $250 \times 10^6 \times 36 \times 10^{-9} = 9$ references. If the probability of a collision is greater than 0, then we need more outstanding references, since we cannot start work on those references; the memory system needs more independent references not fewer! To approximate this, we can simply assume that half the memory references need not be issued to the memory. This means that we must support twice as many outstanding references, or 18.

In Li, Chen, Brockman, and Jouppi's study they found that the reduction in CPI for the integer programs was about 7% for one hit under miss and about 12.7% for 64. For the floating point programs, the reductions were 12.7% for one hit under miss and 17.8% for 64. These reductions track fairly closely the reductions in the data cache access latency shown in Figure 2.5.

Fifth Optimization: Multibanked Caches to Increase Cache Bandwidth

Rather than treat the cache as a single monolithic block, we can divide it into independent banks that can support simultaneous accesses. Banks were originally

Block address	Bank 0	Block address	Bank 1	Block address	Bank 2	Block address	Bank 3
0		1		2		3	
4		5		6		7	
8		9		10		11	
12		13		14		15	

Figure 2.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

used to improve performance of main memory and are now used inside modern DRAM chips as well as with caches. The Arm Cortex-A8 supports one to four banks in its L2 cache; the Intel Core i7 has four banks in L1 (to support up to 2 memory accesses per clock), and the L2 has eight banks.

Clearly, banking works best when the accesses naturally spread themselves across the banks, so the mapping of addresses to banks affects the behavior of the memory system. A simple mapping that works well is to spread the addresses of the block sequentially across the banks, called *sequential interleaving*. For example, if there are four banks, bank 0 has all blocks whose address modulo 4 is 0, bank 1 has all blocks whose address modulo 4 is 1, and so on. Figure 2.6 shows this interleaving. Multiple banks also are a way to reduce power consumption both in caches and DRAM.

Sixth Optimization: Critical Word First and Early Restart to Reduce Miss Penalty

This technique is based on the observation that the processor normally needs just one word of the block at a time. This strategy is impatience: Don't wait for the full block to be loaded before sending the requested word and restarting the processor. Here are two specific strategies:

- *Critical word first*—Request the missed word first from memory and send it to the processor as soon as it arrives; let the processor continue execution while filling the rest of the words in the block.
- *Early restart*—Fetch the words in normal order, but as soon as the requested word of the block arrives send it to the processor and let the processor continue execution.

Generally, these techniques only benefit designs with large cache blocks, since the benefit is low unless blocks are large. Note that caches normally continue to satisfy accesses to other blocks while the rest of the block is being filled.

Alas, given spatial locality, there is a good chance that the next reference is to the rest of the block. Just as with nonblocking caches, the miss penalty is not simple to calculate. When there is a second request in critical word first, the effective miss penalty is the nonoverlapped time from the reference until the

second piece arrives. The benefits of critical word first and early restart depend on the size of the block and the likelihood of another access to the portion of the block that has not yet been fetched.

Seventh Optimization: Merging Write Buffer to Reduce Miss Penalty

Write-through caches rely on write buffers, as all stores must be sent to the next lower level of the hierarchy. Even write-back caches use a simple buffer when a block is replaced. If the write buffer is empty, the data and the full address are written in the buffer, and the write is finished from the processor's perspective; the processor continues working while the write buffer prepares to write the word to memory. If the buffer contains other modified blocks, the addresses can be checked to see if the address of the new data matches the address of a valid write buffer entry. If so, the new data are combined with that entry. *Write merging* is the name of this optimization. The Intel Core i7, among many others, uses write merging.

If the buffer is full and there is no address match, the cache (and processor) must wait until the buffer has an empty entry. This optimization uses the memory more efficiently since multiword writes are usually faster than writes performed one word at a time. Skadron and Clark [1997] found that even a merging four-entry write buffer generated stalls that led to a 5% to 10% performance loss.

The optimization also reduces stalls due to the write buffer being full. Figure 2.7 shows a write buffer with and without write merging. Assume we had four entries in the write buffer, and each entry could hold four 64-bit words. Without this optimization, four stores to sequential addresses would fill the buffer at one word per entry, even though these four words when merged exactly fit within a single entry of the write buffer.

Note that input/output device registers are often mapped into the physical address space. These I/O addresses *cannot* allow write merging because separate I/O registers may not act like an array of words in memory. For example, they may require one address and data word per I/O register rather than use multiword writes using a single address. These side effects are typically implemented by marking the pages as requiring nonmerging write through by the caches.

Eighth Optimization: Compiler Optimizations to Reduce Miss Rate

Thus far, our techniques have required changing the hardware. This next technique reduces miss rates without any hardware changes.

This magical reduction comes from optimized software—the hardware designer's favorite solution! The increasing performance gap between processors and main memory has inspired compiler writers to scrutinize the memory hierarchy to see if compile time optimizations can improve performance. Once again, research

Write address	V	V	V	V
100	1	Mem[100]	0	0
108	1	Mem[108]	0	0
116	1	Mem[116]	0	0
124	1	Mem[124]	0	0

Write address	V	V	V	V
100	1	Mem[100]	1	Mem[108]
	0		0	
	0		0	
	0		0	

Figure 2.7 To illustrate write merging, the write buffer on top does not use it while the write buffer on the bottom does. The four writes are merged into a single buffer entry with write merging; without it, the buffer is full even though three-fourths of each entry is wasted. The buffer has four entries, and each entry holds four 64-bit words. The address for each entry is on the left, with a valid bit (V) indicating whether the next sequential 8 bytes in this entry are occupied. (Without write merging, the words to the right in the upper part of the figure would only be used for instructions that wrote multiple words at the same time.)

is split between improvements in instruction misses and improvements in data misses. The optimizations presented below are found in many modern compilers.

Loop Interchange

Some programs have nested loops that access data in memory in nonsequential order. Simply exchanging the nesting of the loops can make the code access the data in the order in which they are stored. Assuming the arrays do not fit in the cache, this technique reduces misses by improving spatial locality; reordering maximizes use of data in a cache block before they are discarded. For example, if x is a two-dimensional array of size $[5000, 100]$ allocated so that $x[i, j]$ and $x[i, j+1]$ are adjacent (an order called row major, since the array is laid out by rows), then the two pieces of code below show how the accesses can be optimized:

```
/* Before */
for (j = 0; j < 100; j = j+1)
    for (i = 0; i < 5000; i = i+1)
        x[i][j] = 2 * x[i][j];
```

```

/* After */
for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
        x[i][j] = 2 * x[i][j];

```

The original code would skip through memory in strides of 100 words, while the revised version accesses all the words in one cache block before going to the next block. This optimization improves cache performance without affecting the number of instructions executed.

Blocking

This optimization improves temporal locality to reduce misses. We are again dealing with multiple arrays, with some arrays accessed by rows and some by columns. Storing the arrays row by row (*row major order*) or column by column (*column major order*) does not solve the problem because both rows and columns are used in every loop iteration. Such orthogonal accesses mean that transformations such as loop interchange still leave plenty of room for improvement.

Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or *blocks*. The goal is to maximize accesses to the data loaded into the cache before the data are replaced. The code example below, which performs matrix multiplication, helps motivate the optimization:

```

/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        {r = 0;
         for (k = 0; k < N; k = k + 1)
             r = r + y[i][k]*z[k][j];
         x[i][j] = r;
        };

```

The two inner loops read all N -by- N elements of z , read the same N elements in a row of y repeatedly, and write one row of N elements of x . Figure 2.8 gives a snapshot of the accesses to the three arrays. A dark shade indicates a recent access, a light shade indicates an older access, and white means not yet accessed.

The number of capacity misses clearly depends on N and the size of the cache. If it can hold all three N -by- N matrices, then all is well, provided there are no cache conflicts. If the cache can hold one N -by- N matrix and one row of N , then at least the i th row of y and the array z may stay in the cache. Less than that and misses may occur for both x and z . In the worst case, there would be $2N^3 + N^2$ memory words accessed for N^3 operations.

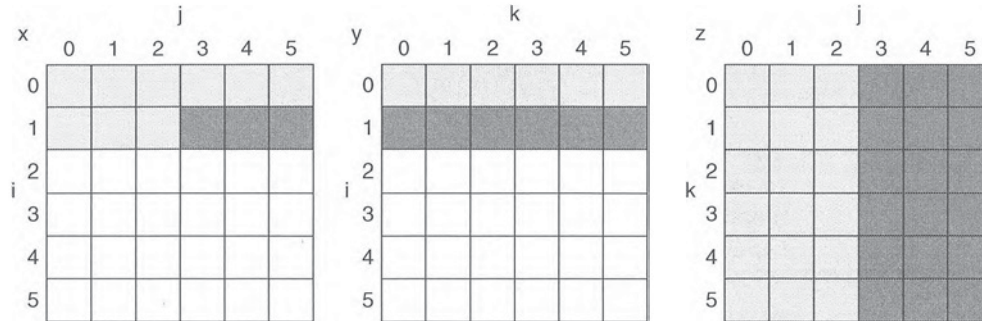


Figure 2.8 A snapshot of the three arrays *x*, *y*, and *z* when $N = 6$ and $i = 1$. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. Compared to Figure 2.9, elements of *y* and *z* are read repeatedly to calculate new elements of *x*. The variables *i*, *j*, and *k* are shown along the rows or columns used to access the arrays.

To ensure that the elements being accessed can fit in the cache, the original code is changed to compute on a submatrix of size B by B . Two inner loops now compute in steps of size B rather than the full length of *x* and *z*. B is called the *blocking factor*. (Assume *x* is initialized to zero.)

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
  for (kk = 0; kk < N; kk = kk+B)
    for (i = 0; i < N; i = i+1)
      for (j = jj; j < min(jj+B,N); j = j+1)
        {r = 0;
         for (k = kk; k < min(kk+B,N); k = k + 1)
           r = r + y[i][k]*z[k][j];
         x[i][j] = x[i][j] + r;
        };
```

Figure 2.9 illustrates the accesses to the three arrays using blocking. Looking only at capacity misses, the total number of memory words accessed is $2N^3/B + N^2$. This total is an improvement by about a factor of B . Hence, blocking exploits a combination of spatial and temporal locality, since *y* benefits from spatial locality and *z* benefits from temporal locality.

Although we have aimed at reducing cache misses, blocking can also be used to help register allocation. By taking a small blocking size such that the block can be held in registers, we can minimize the number of loads and stores in the program.

As we shall see in Section 4.8 of Chapter 4, cache blocking is absolutely necessary to get good performance from cache-based processors running applications using matrices as the primary data structure.

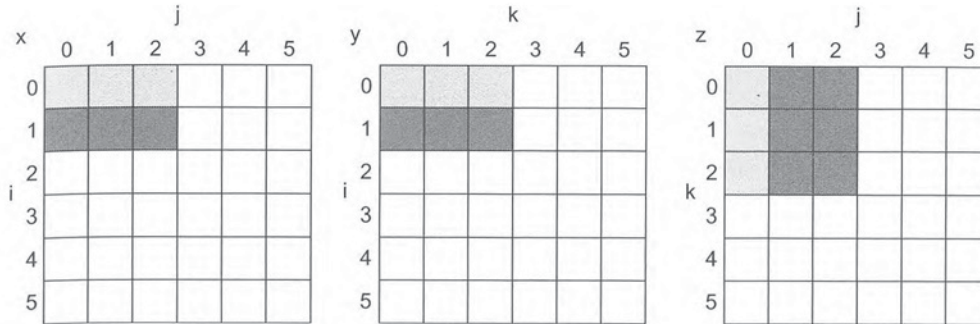


Figure 2.9 The age of accesses to the arrays x , y , and z when $B = 3$. Note that, in contrast to Figure 2.8, a smaller number of elements is accessed.

Ninth Optimization: Hardware Prefetching of Instructions and Data to Reduce Miss Penalty or Miss Rate

Nonblocking caches effectively reduce the miss penalty by overlapping execution with memory access. Another approach is to prefetch items before the processor requests them. Both instructions and data can be prefetched, either directly into the caches or into an external buffer that can be more quickly accessed than main memory.

Instruction prefetch is frequently done in hardware outside of the cache. Typically, the processor fetches two blocks on a miss: the requested block and the next consecutive block. The requested block is placed in the instruction cache when it returns, and the prefetched block is placed into the instruction stream buffer. If the requested block is present in the instruction stream buffer, the original cache request is canceled, the block is read from the stream buffer, and the next prefetch request is issued.

A similar approach can be applied to data accesses [Jouppi 1990]. Palacharla and Kessler [1994] looked at a set of scientific programs and considered multiple stream buffers that could handle either instructions or data. They found that eight stream buffers could capture 50% to 70% of all misses from a processor with two 64 KB four-way set associative caches, one for instructions and the other for data.

The Intel Core i7 supports hardware prefetching into both L1 and L2 with the most common case of prefetching being accessing the next line. Some earlier Intel processors used more aggressive hardware prefetching, but that resulted in reduced performance for some applications, causing some sophisticated users to turn off the capability.

Figure 2.10 shows the overall performance improvement for a subset of SPEC2000 programs when hardware prefetching is turned on. Note that this figure includes only 2 of 12 integer programs, while it includes the majority of the SPEC floating-point programs.

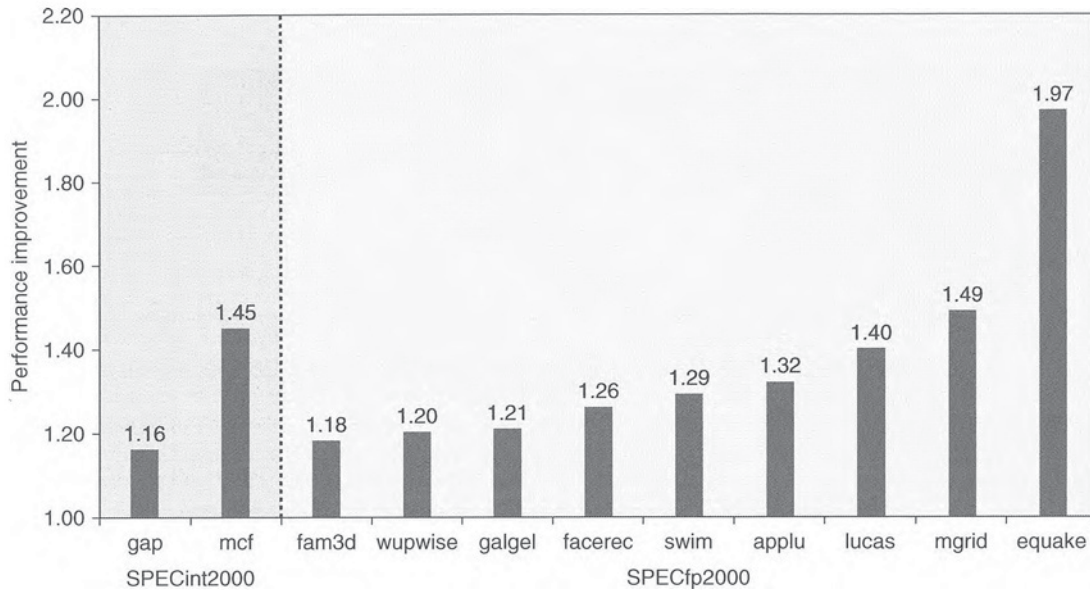


Figure 2.10 Speedup due to hardware prefetching on Intel Pentium 4 with hardware prefetching turned on for 2 of 12 SPECint2000 benchmarks and 9 of 14 SPECfp2000 benchmarks. Only the programs that benefit the most from prefetching are shown; prefetching speeds up the missing 15 SPEC benchmarks by less than 15% [Singhal 2004].

Prefetching relies on utilizing memory bandwidth that otherwise would be unused, but if it interferes with demand misses it can actually lower performance. Help from compilers can reduce useless prefetching. When prefetching works well its impact on power is negligible. When prefetched data are not used or useful data are displaced, prefetching will have a very negative impact on power.

Tenth Optimization: Compiler-Controlled Prefetching to Reduce Miss Penalty or Miss Rate

An alternative to hardware prefetching is for the compiler to insert prefetch instructions to request data before the processor needs it. There are two flavors of prefetch:

- *Register prefetch* will load the value into a register.
- *Cache prefetch* loads data only into the cache and not the register.

Either of these can be *faulting* or *nonfaulting*; that is, the address does or does not cause an exception for virtual address faults and protection violations. Using this terminology, a normal load instruction could be considered a “faulting register prefetch instruction.” Nonfaulting prefetches simply turn into no-ops if they would normally result in an exception, which is what we want.

The most effective prefetch is “semantically invisible” to a program: It doesn’t change the contents of registers and memory, *and* it cannot cause virtual memory faults. Most processors today offer nonfaulting cache prefetches. This section assumes nonfaulting cache prefetch, also called *nonbinding* prefetch.

Prefetching makes sense only if the processor can proceed while prefetching the data; that is, the caches do not stall but continue to supply instructions and data while waiting for the prefetched data to return. As you would expect, the data cache for such computers is normally nonblocking.

Like hardware-controlled prefetching, the goal is to overlap execution with the prefetching of data. Loops are the important targets, as they lend themselves to prefetch optimizations. If the miss penalty is small, the compiler just unrolls the loop once or twice, and it schedules the prefetches with the execution. If the miss penalty is large, it uses software pipelining (see Appendix H) or unrolls many times to prefetch data for a future iteration.

Issuing prefetch instructions incurs an instruction overhead, however, so compilers must take care to ensure that such overheads do not exceed the benefits. By concentrating on references that are likely to be cache misses, programs can avoid unnecessary prefetches while improving average memory access time significantly.

Example For the code below, determine which accesses are likely to cause data cache misses. Next, insert prefetch instructions to reduce misses. Finally, calculate the number of prefetch instructions executed and the misses avoided by prefetching. Let’s assume we have an 8 KB direct-mapped data cache with 16-byte blocks, and it is a write-back cache that does write allocate. The elements of *a* and *b* are 8 bytes long since they are double-precision floating-point arrays. There are 3 rows and 100 columns for *a* and 101 rows and 3 columns for *b*. Let’s also assume they are not in the cache at the start of the program.

```
for (i = 0; i < 3; i = i+1)
    for (j = 0; j < 100; j = j+1)
        a[i][j] = b[j][0] * b[j+1][0];
```

Answer The compiler will first determine which accesses are likely to cause cache misses; otherwise, we will waste time on issuing prefetch instructions for data that would be hits. Elements of *a* are written in the order that they are stored in memory, so *a* will benefit from spatial locality: The even values of *j* will miss and the odd values will hit. Since *a* has 3 rows and 100 columns, its accesses will lead to $3 \times (100/2)$, or 150 misses.

The array *b* does not benefit from spatial locality since the accesses are not in the order it is stored. The array *b* does benefit twice from temporal locality: The same elements are accessed for each iteration of *i*, and each iteration of *j* uses the same value of *b* as the last iteration. Ignoring potential conflict misses, the misses due to *b* will be for *b*[*j*+1][0] accesses when *i* = 0, and also the first

access to $b[j][0]$ when $j = 0$. Since j goes from 0 to 99 when $i = 0$, accesses to b lead to $100 + 1$, or 101 misses.

Thus, this loop will miss the data cache approximately 150 times for a plus 101 times for b , or 251 misses.

To simplify our optimization, we will not worry about prefetching the first accesses of the loop. These may already be in the cache, or we will pay the miss penalty of the first few elements of a or b . Nor will we worry about suppressing the prefetches at the end of the loop that try to prefetch beyond the end of a ($a[i][100] \dots a[i][106]$) and the end of b ($b[101][0] \dots b[107][0]$). If these were faulting prefetches, we could not take this luxury. Let's assume that the miss penalty is so large we need to start prefetching at least, say, seven iterations in advance. (Stated alternatively, we assume prefetching has no benefit until the eighth iteration.) We underline the changes to the code above needed to add prefetching.

```

for (j = 0; j < 100; j = j+1) {
    prefetch(b[j+7][0]);
    /* b(j,0) for 7 iterations later */
    prefetch(a[0][j+7]);
    /* a(0,j) for 7 iterations later */
    a[0][j] = b[j][0] * b[j+1][0];
}
for (i = 1; i < 3; i = i+1)
    for (j = 0; j < 100; j = j+1) {
        prefetch(a[i][j+7]);
        /* a(i,j) for +7 iterations */
        a[i][j] = b[j][0] * b[j+1][0];
    }

```

This revised code prefetches $a[i][7]$ through $a[i][99]$ and $b[7][0]$ through $b[100][0]$, reducing the number of nonprefetched misses to

- 7 misses for elements $b[0][0], b[1][0], \dots, b[6][0]$ in the first loop
- 4 misses ($\lceil 7/2 \rceil$) for elements $a[0][0], a[0][1], \dots, a[0][6]$ in the first loop (spatial locality reduces misses to 1 per 16-byte cache block)
- 4 misses ($\lceil 7/2 \rceil$) for elements $a[1][0], a[1][1], \dots, a[1][6]$ in the second loop
- 4 misses ($\lceil 7/2 \rceil$) for elements $a[2][0], a[2][1], \dots, a[2][6]$ in the second loop

or a total of 19 nonprefetched misses. The cost of avoiding 232 cache misses is executing 400 prefetch instructions, likely a good trade-off.

Example Calculate the time saved in the example above. Ignore instruction cache misses and assume there are no conflict or capacity misses in the data cache. Assume that prefetches can overlap with each other and with cache misses, thereby

transferring at the maximum memory bandwidth. Here are the key loop times ignoring cache misses: The original loop takes 7 clock cycles per iteration, the first prefetch loop takes 9 clock cycles per iteration, and the second prefetch loop takes 8 clock cycles per iteration (including the overhead of the outer for loop). A miss takes 100 clock cycles.

Answer The original doubly nested loop executes the multiply 3×100 or 300 times. Since the loop takes 7 clock cycles per iteration, the total is 300×7 or 2100 clock cycles plus cache misses. Cache misses add 251×100 or 25,100 clock cycles, giving a total of 27,200 clock cycles. The first prefetch loop iterates 100 times; at 9 clock cycles per iteration the total is 900 clock cycles plus cache misses. Now add 11×100 or 1100 clock cycles for cache misses, giving a total of 2000. The second loop executes 2×100 or 200 times, and at 8 clock cycles per iteration it takes 1600 clock cycles plus 8×100 or 800 clock cycles for cache misses. This gives a total of 2400 clock cycles. From the prior example, we know that this code executes 400 prefetch instructions during the $2000 + 2400$ or 4400 clock cycles to execute these two loops. If we assume that the prefetches are completely overlapped with the rest of the execution, then the prefetch code is $27,200/4400$, or 6.2 times faster.

Although array optimizations are easy to understand, modern programs are more likely to use pointers. Luk and Mowry [1999] have demonstrated that compiler-based prefetching can sometimes be extended to pointers as well. Of 10 programs with recursive data structures, prefetching all pointers when a node is visited improved performance by 4% to 31% in half of the programs. On the other hand, the remaining programs were still within 2% of their original performance. The issue is both whether prefetches are to data already in the cache and whether they occur early enough for the data to arrive by the time it is needed.

Many processors support instructions for cache prefetch, and high-end processors (such as the Intel Core i7) often also do some type of automated prefetch in hardware.

Cache Optimization Summary

The techniques to improve hit time, bandwidth, miss penalty, and miss rate generally affect the other components of the average memory access equation as well as the complexity of the memory hierarchy. Figure 2.11 summarizes these techniques and estimates the impact on complexity, with + meaning that the technique improves the factor, – meaning it hurts that factor, and blank meaning it has no impact. Generally, no technique helps more than one category.

Technique	Hit time	Bandwidth	Miss penalty	Miss rate	Power consumption	Hardware cost/complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined cache access	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Banked caches		+			+	1	Used in L2 of both i7 and Cortex-A8
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware.
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs

Figure 2.11 Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity. Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

2.3

Memory Technology and Optimizations

... the one single development that put computers on their feet was the invention of a reliable form of memory, namely, the core memory. ... Its cost was reasonable, it was reliable and, because it was reliable, it could in due course be made large.
[p. 209]

Maurice Wilkes

Memoirs of a Computer Pioneer (1985)

Main memory is the next level down in the hierarchy. Main memory satisfies the demands of caches and serves as the I/O interface, as it is the destination of input as well as the source for output. Performance measures of main memory emphasize both latency and bandwidth. Traditionally, main memory latency (which

affects the cache miss penalty) is the primary concern of the cache, while main memory bandwidth is the primary concern of multiprocessors and I/O.

Although caches benefit from low-latency memory, it is generally easier to improve memory bandwidth with new organizations than it is to reduce latency. The popularity of multilevel caches and their larger block sizes make main memory bandwidth important to caches as well. In fact, cache designers increase block size to take advantage of the high memory bandwidth.

The previous sections describe what can be done with cache organization to reduce this processor–DRAM performance gap, but simply making caches larger or adding more levels of caches cannot eliminate the gap. Innovations in main memory are needed as well.

In the past, the innovation was how to organize the many DRAM chips that made up the main memory, such as multiple memory banks. Higher bandwidth is available using memory banks, by making memory and its bus wider, or by doing both. Ironically, as capacity per memory chip increases, there are fewer chips in the same-sized memory system, reducing possibilities for wider memory systems with the same capacity.

To allow memory systems to keep up with the bandwidth demands of modern processors, memory innovations started happening inside the DRAM chips themselves. This section describes the technology inside the memory chips and those innovative, internal organizations. Before describing the technologies and options, let's go over the performance metrics.

With the introduction of burst transfer memories, now widely used in both Flash and DRAM, memory latency is quoted using two measures—access time and cycle time. *Access time* is the time between when a read is requested and when the desired word arrives, and *cycle time* is the minimum time between unrelated requests to memory.

Virtually all computers since 1975 have used DRAMs for main memory and SRAMs for cache, with one to three levels integrated onto the processor chip with the CPU. In PMDs, the memory technology often balances power and speed, with higher end systems using fast, high-bandwidth memory technology.

SRAM Technology

The first letter of SRAM stands for *static*. The dynamic nature of the circuits in DRAM requires data to be written back after being read—hence the difference between the access time and the cycle time as well as the need to refresh. SRAMs don't need to refresh, so the access time is very close to the cycle time. SRAMs typically use six transistors per bit to prevent the information from being disturbed when read. SRAM needs only minimal power to retain the charge in standby mode.

In earlier times, most desktop and server systems used SRAM chips for their primary, secondary, or tertiary caches; today, all three levels of caches are integrated onto the processor chip. Currently, the largest on-chip, third-level caches are 12 MB, while the memory system for such a processor is likely to have 4 to

16 GB of DRAM. The access times for large, third-level, on-chip caches are typically two to four times that of a second-level cache, which is still three to five times faster than accessing DRAM memory.

DRAM Technology

As early DRAMs grew in capacity, the cost of a package with all the necessary address lines was an issue. The solution was to multiplex the address lines, thereby cutting the number of address pins in half. Figure 2.12 shows the basic DRAM organization. One-half of the address is sent first during the *row access strobe* (RAS). The other half of the address, sent during the *column access strobe* (CAS), follows it. These names come from the internal chip organization, since the memory is organized as a rectangular matrix addressed by rows and columns.

An additional requirement of DRAM derives from the property signified by its first letter, *D*, for *dynamic*. To pack more bits per chip, DRAMs use only a single transistor to store a bit. Reading that bit destroys the information, so it must be restored. This is one reason why the DRAM cycle time was traditionally longer than the access time; more recently, DRAMs have introduced multiple banks, which allow the rewrite portion of the cycle to be hidden. In addition, to prevent loss of information when a bit is not read or written, the bit must be “refreshed” periodically. Fortunately, all the bits in a row can be refreshed simultaneously just by reading that row. Hence, every DRAM in the memory system must access every row within a certain time window, such as 8 ms. Memory controllers include hardware to refresh the DRAMs periodically.

This requirement means that the memory system is occasionally unavailable because it is sending a signal telling every chip to refresh. The time for a refresh is typically a full memory access (RAS and CAS) for each row of the DRAM. Since the memory matrix in a DRAM is conceptually square, the number of steps

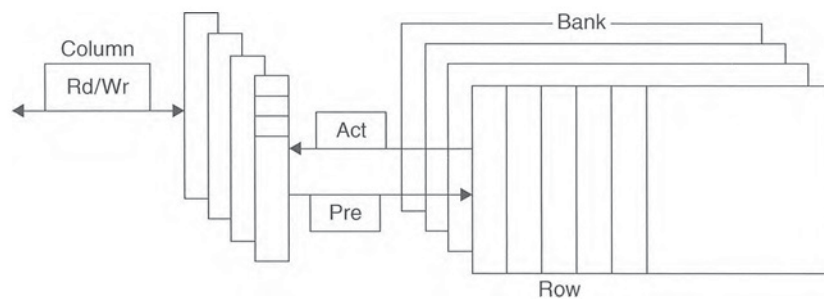


Figure 2.12 Internal organization of a DRAM. Modern DRAMs are organized in banks, typically four for DDR3. Each bank consists of a series of rows. Sending a PRE (pre-charge) command opens or closes a bank. A row address is sent with an Act (activate), which causes the row to transfer to a buffer. When the row is in the buffer, it can be transferred by successive column addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR3) or by specifying a block transfer and the starting address. Each command, as well as block transfers, are synchronized with a clock.

in a refresh is usually the square root of the DRAM capacity. DRAM designers try to keep time spent refreshing to less than 5% of the total time.

So far we have presented main memory as if it operated like a Swiss train, consistently delivering the goods exactly according to schedule. Refresh belies that analogy, since some accesses take much longer than others do. Thus, refresh is another reason for variability of memory latency and hence cache miss penalty.

Amdahl suggested as a rule of thumb that memory capacity should grow linearly with processor speed to keep a balanced system, so that a 1000 MIPS processor should have 1000 MB of memory. Processor designers rely on DRAMs to supply that demand. In the past, they expected a fourfold improvement in capacity every three years, or 55% per year. Unfortunately, the performance of DRAMs is growing at a much slower rate. Figure 2.13 shows a performance improvement in row access time, which is related to latency, of about 5% per year. The CAS or data transfer time, which is related to bandwidth, is growing at more than twice that rate.

Although we have been talking about individual chips, DRAMs are commonly sold on small boards called *dual inline memory modules* (DIMMs). DIMMs typically contain 4 to 16 DRAMs, and they are normally organized to be 8 bytes wide (+ ECC) for desktop and server systems.

Production year	Chip size	DRAM type	Row access strobe (RAS)		Column access strobe (CAS)/ data transfer time (ns)	Cycle time (ns)
			Slowest DRAM (ns)	Fastest DRAM (ns)		
1980	64K bit	DRAM	180	150	75	250
1983	256K bit	DRAM	150	120	50	220
1986	1M bit	DRAM	120	100	25	190
1989	4M bit	DRAM	100	80	20	165
1992	16M bit	DRAM	80	60	15	120
1996	64M bit	SDRAM	70	50	12	110
1998	128M bit	SDRAM	70	50	10	100
2000	256M bit	DDR1	65	45	7	90
2002	512M bit	DDR1	60	40	5	80
2004	1G bit	DDR2	55	35	5	70
2006	2G bit	DDR2	50	30	2.5	60
2010	4G bit	DDR3	36	28	1	37
2012	8G bit	DDR3	30	24	0.5	31

Figure 2.13 Times of fast and slow DRAMs vary with each generation. (Cycle time is defined on page 97.) Performance improvement of row access time is about 5% per year. The improvement by a factor of 2 in column access in 1986 accompanied the switch from NMOS DRAMs to CMOS DRAMs. The introduction of various burst transfer modes in the mid-1990s and SDRAMs in the late 1990s has significantly complicated the calculation of access time for blocks of data; we discuss this later in this section when we talk about SDRAM access time and power. The DDR4 designs are due for introduction in mid- to late 2012. We discuss these various forms of DRAMs in the next few pages.

In addition to the DIMM packaging and the new interfaces to improve the data transfer time, discussed in the following subsections, the biggest change to DRAMs has been a slowing down in capacity growth. DRAMs obeyed Moore's law for 20 years, bringing out a new chip with four times the capacity every three years. Due to the manufacturing challenges of a single-bit DRAM, new chips only double capacity every two years since 1998. In 2006, the pace slowed further, with the four years from 2006 to 2010 seeing only a doubling of capacity.

Improving Memory Performance Inside a DRAM Chip

As Moore's law continues to supply more transistors and as the processor-memory gap increases pressure on memory performance, the ideas of the previous section have made their way inside the DRAM chip. Generally, innovation has led to greater bandwidth, sometimes at the cost of greater latency. This subsection presents techniques that take advantage of the nature of DRAMs.

As mentioned earlier, a DRAM access is divided into row access and column access. DRAMs must buffer a row of bits inside the DRAM for the column access, and this row is usually the square root of the DRAM size—for example, 2 Kb for a 4 Mb DRAM. As DRAMs grew, additional structure and several opportunities for increasing bandwidth were added.

First, DRAMs added timing signals that allow repeated accesses to the row buffer without another row access time. Such a buffer comes naturally, as each array will buffer 1024 to 4096 bits for each access. Initially, separate column addresses had to be sent for each transfer with a delay after each new set of column addresses.

Originally, DRAMs had an asynchronous interface to the memory controller, so every transfer involved overhead to synchronize with the controller. The second major change was to add a clock signal to the DRAM interface, so that the repeated transfers would not bear that overhead. *Synchronous DRAM* (SDRAM) is the name of this optimization. SDRAMs typically also have a programmable register to hold the number of bytes requested, and hence can send many bytes over several cycles per request. Typically, 8 or more 16-bit transfers can occur without sending any new addresses by placing the DRAM in burst mode; this mode, which supports critical word first transfers, is the only way that the peak bandwidths shown in Figure 2.14 can be achieved.

Third, to overcome the problem of getting a wide stream of bits from the memory without having to make the memory system too large as memory system density increased, DRAMs were made wider. Initially, they offered a four-bit transfer mode; in 2010, DDR2 and DDR3 DRAMs had up to 16-bit buses.

The fourth major DRAM innovation to increase bandwidth is to transfer data on both the rising edge and falling edge of the DRAM clock signal, thereby doubling the peak data rate. This optimization is called *double data rate* (DDR).

To provide some of the advantages of interleaving, as well to help with power management, SDRAMs also introduced *banks*, breaking a single SDRAM into 2 to 8 blocks (in current DDR3 DRAMs) that can operate independently. (We have already seen banks used in internal caches, and they were often used in large

Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1066–1600	2133–3200	DDR4-3200	17,056–25,600	PC25600

Figure 2.14 Clock rates, bandwidth, and names of DDR DRAMS and DIMMs in 2010. Note the numerical relationship between the columns. The third column is twice the second, and the fourth uses the number from the third column in the name of the DRAM chip. The fifth column is eight times the third column, and a rounded version of this number is used in the name of the DIMM. Although not shown in this figure, DDRs also specify latency in clock cycles as four numbers, which are specified by the DDR standard. For example, DDR3-2000 CL 9 has latencies of 9-9-9-28. What does this mean? With a 1 ns clock (clock cycle is one-half the transfer rate), this indicates 9 ns for row to columns address (RAS time), 9 ns for column access to data (CAS time), and a minimum read time of 28 ns. Closing the row takes 9 ns for precharge but happens only when the reads from that row are finished. In burst mode, transfers occur on every clock on both edges, when the first RAS and CAS times have elapsed. Furthermore, the precharge is not needed until the entire row is read. DDR4 will be produced in 2012 and is expected to reach clock rates of 1600 MHz in 2014, when DDR5 is expected to take over. The exercises explore these details further.

main memories.) Creating multiple banks inside a DRAM effectively adds another segment to the address, which now consists of bank number, row address, and column address. When an address is sent that designates a new bank, that bank must be opened, incurring an additional delay. The management of banks and row buffers is completely handled by modern memory control interfaces, so that when subsequent access specifies the same row for an open bank, the access can happen quickly, sending only the column address.

When DDR SDRAMs are packaged as DIMMs, they are confusingly labeled by the peak *DIMM* bandwidth. Hence, the DIMM name PC2100 comes from $133 \text{ MHz} \times 2 \times 8 \text{ bytes}$, or 2100 MB/sec. Sustaining the confusion, the chips themselves are labeled with *the number of bits per second* rather than their clock rate, so a 133 MHz DDR chip is called a DDR266. Figure 2.14 shows the relationships among clock rate, transfers per second per chip, chip name, DIMM bandwidth, and DIMM name.

DDR is now a sequence of standards. DDR2 lowers power by dropping the voltage from 2.5 volts to 1.8 volts and offers higher clock rates: 266 MHz, 333 MHz, and 400 MHz. DDR3 drops voltage to 1.5 volts and has a maximum clock speed of 800 MHz. DDR4, scheduled for production in 2014, drops the voltage to 1 to 1.2 volts and has a maximum expected clock rate of 1600 MHz. DDR5 will follow in about 2014 or 2015. (As we discuss in the next section, GDDR5 is a graphics RAM and is based on DDR3 DRAMs.)

Graphics Data RAMs

GDRAMs or GSDRAMs (Graphics or Graphics Synchronous DRAMs) are a special class of DRAMs based on SDRAM designs but tailored for handling the higher bandwidth demands of graphics processing units. GDDR5 is based on DDR3 with earlier GDDRs based on DDR2. Since Graphics Processor Units (GPUs; see Chapter 4) require more bandwidth per DRAM chip than CPUs, GDDRs have several important differences:

1. GDDRs have wider interfaces: 32-bits versus 4, 8, or 16 in current designs.
2. GDDRs have a higher maximum clock rate on the data pins. To allow a higher transfer rate without incurring signaling problems, GDRAMs normally connect directly to the GPU and are attached by soldering them to the board, unlike DRAMs, which are normally arranged in an expandable array of DIMMs.

Altogether, these characteristics let GDDRs run at two to five times the bandwidth per DRAM versus DDR3 DRAMs, a significant advantage in supporting GPUs. Because of the lower locality of memory requests in a GPU, burst mode generally is less useful for a GPU, but keeping open multiple memory banks and managing their use improves effective bandwidth.

Reducing Power Consumption in SDRAMs

Power consumption in dynamic memory chips consists of both dynamic power used in a read or write and static or standby power; both depend on the operating voltage. In the most advanced DDR3 SDRAMs the operating voltage has been dropped to 1.35 to 1.5 volts, significantly reducing power versus DDR2 SDRAMs. The addition of banks also reduced power, since only the row in a single bank is read and precharged.

In addition to these changes, all recent SDRAMs support a power down mode, which is entered by telling the DRAM to ignore the clock. Power down mode disables the SDRAM, except for internal automatic refresh (without which entering power down mode for longer than the refresh time will cause the contents of memory to be lost). Figure 2.15 shows the power consumption for three situations in a 2 Gb DDR3 SDRAM. The exact delay required to return from low power mode depends on the SDRAM, but a typical timing from autorefresh low power mode is 200 clock cycles; additional time may be required for resetting the mode register before the first command.

Flash Memory

Flash memory is a type of EEPROM (Electrically Erasable Programmable Read-Only Memory), which is normally read-only but can be erased. The other key property of Flash memory is that it holds its contents without any power.

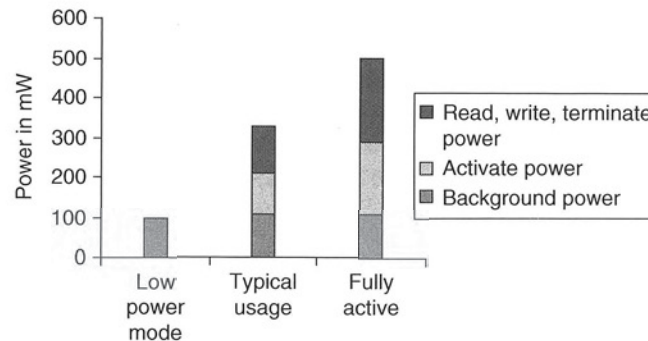


Figure 2.15 Power consumption for a DDR3 SDRAM operating under three conditions: low power (shutdown) mode, typical system mode (DRAM is active 30% of the time for reads and 15% for writes), and fully active mode, where the DRAM is continuously reading or writing when not in precharge. Reads and writes assume bursts of 8 transfers. These data are based on a Micron 1.5V 2Gb DDR3-1066.

Flash is used as the backup storage in PMDs in the same manner that a disk functions in a laptop or server. In addition, because most PMDs have a limited amount of DRAM, Flash may also act as a level of the memory hierarchy, to a much larger extent than it might have to do so in the desktop or server with a main memory that might be 10 to 100 times larger.

Flash uses a very different architecture and has different properties than standard DRAM. The most important differences are

1. Flash memory must be erased (hence the name Flash for the “flash” erase process) before it is overwritten, and it is erased in blocks (in high-density Flash, called NAND Flash, which is what is used in most computer applications) rather than individual bytes or words. This means when data must be written to Flash, an entire block must be assembled, either as new data or by merging the data to be written and the rest of the block’s contents.
2. Flash memory is static (i.e., it keeps its contents even when power is not applied) and draws significantly less power when not reading or writing (from less than half in standby mode to zero when completely inactive).
3. Flash memory has a limited number of write cycles for any block, typically at least 100,000. By ensuring uniform distribution of written blocks throughout the memory, a system can maximize the lifetime of a Flash memory system.
4. High-density Flash is cheaper than SDRAM but more expensive than disks: roughly \$2/GB for Flash, \$20 to \$40/GB for SDRAM, and \$0.09/GB for magnetic disks.
5. Flash is much slower than SDRAM but much faster than disk. For example, a transfer of 256 bytes from a typical high-density Flash memory takes about 6.5 μ s (using burst mode transfer similar to but slower than that used in SDRAM). A comparable transfer from a DDR SDRAM takes about one-quarter as long, and for a disk about 1000 times longer. For writes, the

difference is considerably larger, with the SDRAM being at least 10 and as much as 100 times faster than Flash depending on the circumstances.

The rapid improvements in high-density Flash in the past decade have made the technology a viable part of memory hierarchies in mobile devices and as solid-state replacements for disks. As the rate of increase in DRAM density continues to drop, Flash could play an increased role in future memory systems, acting as both a replacement for hard disks and as an intermediate storage between DRAM and disk.

Enhancing Dependability in Memory Systems

Large caches and main memories significantly increase the possibility of errors occurring both during the fabrication process and dynamically, primarily from cosmic rays striking a memory cell. These dynamic errors, which are changes to a cell's contents, not a change in the circuitry, are called *soft errors*. All DRAMs, Flash memory, and many SRAMs are manufactured with spare rows, so that a small number of manufacturing defects can be accommodated by programming the replacement of a defective row by a spare row. In addition to fabrication errors that must be fixed at configuration time, *hard errors*, which are permanent changes in the operation of one of more memory cells, can occur in operation.

Dynamic errors can be detected by parity bits and detected and fixed by the use of Error Correcting Codes (ECCs). Because instruction caches are read-only, parity suffices. In larger data caches and in main memory, ECC is used to allow errors to be both detected and corrected. Parity requires only one bit of overhead to detect a single error in a sequence of bits. Because a multibit error would be undetected with parity, the number of bits protected by a parity bit must be limited. One parity bit per 8 data bits is a typical ratio. ECC can detect two errors and correct a single error with a cost of 8 bits of overhead per 64 data bits.

In very large systems, the possibility of multiple errors as well as complete failure of a single memory chip becomes significant. Chipkill was introduced by IBM to solve this problem, and many very large systems, such as IBM and SUN servers and the Google Clusters, use this technology. (Intel calls their version SDDC.) Similar in nature to the RAID approach used for disks, Chipkill distributes the data and ECC information, so that the complete failure of a single memory chip can be handled by supporting the reconstruction of the missing data from the remaining memory chips. Using an analysis by IBM and assuming a 10,000 processor server with 4 GB per processor yields the following rates of unrecoverable errors in three years of operation:

- Parity only—about 90,000, or one unrecoverable (or undetected) failure every 17 minutes
- ECC only—about 3500, or about one undetected or unrecoverable failure every 7.5 hours
- Chipkill—6, or about one undetected or unrecoverable failure every 2 months

Another way to look at this is to find the maximum number of servers (each with 4 GB) that can be protected while achieving the same error rate as demonstrated for Chipkill. For parity, even a server with only one processor will have an unrecoverable error rate higher than a 10,000-server Chipkill protected system. For ECC, a 17-server system would have about the same failure rate as a 10,000-server Chipkill system. Hence, Chipkill is a requirement for the 50,000 to 100,00 servers in warehouse-scale computers (see Section 6.8 of Chapter 6).

2.4

Protection: Virtual Memory and Virtual Machines

A virtual machine is taken to be an efficient, isolated duplicate of the real machine. We explain these notions through the idea of a virtual machine monitor (VMM). . . . a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.

Gerald Popek and Robert Goldberg

"Formal requirements for virtualizable third generation architectures,"

Communications of the ACM (July 1974)

Security and privacy are two of the most vexing challenges for information technology in 2011. Electronic burglaries, often involving lists of credit card numbers, are announced regularly, and it's widely believed that many more go unreported. Hence, both researchers and practitioners are looking for new ways to make computing systems more secure. Although protecting information is not limited to hardware, in our view real security and privacy will likely involve innovation in computer architecture as well as in systems software.

This section starts with a review of the architecture support for protecting processes from each other via virtual memory. It then describes the added protection provided from virtual machines, the architecture requirements of virtual machines, and the performance of a virtual machine. As we will see in Chapter 6, virtual machines are a foundational technology for cloud computing.

Protection via Virtual Memory

Page-based virtual memory, including a translation lookaside buffer that caches page table entries, is the primary mechanism that protects processes from each other. Sections B.4 and B.5 in Appendix B review virtual memory, including a detailed description of protection via segmentation and paging in the 80x86. This subsection acts as a quick review; refer to those sections if it's too quick.

Multiprogramming, where several programs running concurrently would share a computer, led to demands for protection and sharing among programs and

to the concept of a *process*. Metaphorically, a process is a program's breathing air and living space—that is, a running program plus any state needed to continue running it. At any instant, it must be possible to switch from one process to another. This exchange is called a *process switch* or *context switch*.

The operating system and architecture join forces to allow processes to share the hardware yet not interfere with each other. To do this, the architecture must limit what a process can access when running a user process yet allow an operating system process to access more. At a minimum, the architecture must do the following:

1. Provide at least two modes, indicating whether the running process is a user process or an operating system process. This latter process is sometimes called a *kernel* process or a *supervisor* process.
2. Provide a portion of the processor state that a user process can use but not write. This state includes a user/supervisor mode bit, an exception enable/disable bit, and memory protection information. Users are prevented from writing this state because the operating system cannot control user processes if users can give themselves supervisor privileges, disable exceptions, or change memory protection.
3. Provide mechanisms whereby the processor can go from user mode to supervisor mode and vice versa. The first direction is typically accomplished by a *system call*, implemented as a special instruction that transfers control to a dedicated location in supervisor code space. The PC is saved from the point of the system call, and the processor is placed in supervisor mode. The return to user mode is like a subroutine return that restores the previous user/supervisor mode.
4. Provide mechanisms to limit memory accesses to protect the memory state of a process without having to swap the process to disk on a context switch.

Appendix A describes several memory protection schemes, but by far the most popular is adding protection restrictions to each page of virtual memory. Fixed-sized pages, typically 4 KB or 8 KB long, are mapped from the virtual address space into physical address space via a page table. The protection restrictions are included in each page table entry. The protection restrictions might determine whether a user process can read this page, whether a user process can write to this page, and whether code can be executed from this page. In addition, a process can neither read nor write a page if it is not in the page table. Since only the OS can update the page table, the paging mechanism provides total access protection.

Paged virtual memory means that every memory access logically takes at least twice as long, with one memory access to obtain the physical address and a second access to get the data. This cost would be far too dear. The solution is to rely on the principle of locality; if the accesses have locality, then the *address translations* for the accesses must also have locality. By keeping these address translations in a special cache, a memory access rarely requires a second access to translate the address. This special address translation cache is referred to as a *translation lookaside buffer* (TLB).

A TLB entry is like a cache entry where the tag holds portions of the virtual address and the data portion holds a physical page address, protection field, valid bit, and usually a use bit and a dirty bit. The operating system changes these bits by changing the value in the page table and then invalidating the corresponding TLB entry. When the entry is reloaded from the page table, the TLB gets an accurate copy of the bits.

Assuming the computer faithfully obeys the restrictions on pages and maps virtual addresses to physical addresses, it would seem that we are done. Newspaper headlines suggest otherwise.

The reason we're not done is that we depend on the accuracy of the operating system as well as the hardware. Today's operating systems consist of tens of millions of lines of code. Since bugs are measured in number per thousand lines of code, there are thousands of bugs in production operating systems. Flaws in the OS have led to vulnerabilities that are routinely exploited.

This problem and the possibility that not enforcing protection could be much more costly than in the past have led some to look for a protection model with a much smaller code base than the full OS, such as Virtual Machines.

Protection via Virtual Machines

An idea related to virtual memory that is almost as old are Virtual Machines (VMs). They were first developed in the late 1960s, and they have remained an important part of mainframe computing over the years. Although largely ignored in the domain of single-user computers in the 1980s and 1990s, they have recently gained popularity due to

- The increasing importance of isolation and security in modern systems
- The failures in security and reliability of standard operating systems
- The sharing of a single computer among many unrelated users, such as in a datacenter or cloud
- The dramatic increases in the raw speed of processors, which make the overhead of VMs more acceptable

The broadest definition of VMs includes basically all emulation methods that provide a standard software interface, such as the Java VM. We are interested in VMs that provide a complete system-level environment at the binary instruction set architecture (ISA) level. Most often, the VM supports the same ISA as the underlying hardware; however, it is also possible to support a different ISA, and such approaches are often employed when migrating between ISAs, so as to allow software from the departing ISA to be used until it can be ported to the new ISA. Our focus here will be in VMs where the ISA presented by the VM and the underlying hardware match. Such VMs are called (Operating) *System Virtual Machines*. IBM VM/370, VMware ESX Server, and Xen are examples. They present the illusion that the users of a VM have an entire computer to themselves,

including a copy of the operating system. A single computer runs multiple VMs and can support a number of different operating systems (OSes). On a conventional platform, a single OS “owns” all the hardware resources, but with a VM multiple OSes all share the hardware resources.

The software that supports VMs is called a *virtual machine monitor* (VMM) or *hypervisor*; the VMM is the heart of virtual machine technology. The underlying hardware platform is called the *host*, and its resources are shared among the *guest* VMs. The VMM determines how to map virtual resources to physical resources: A physical resource may be time-shared, partitioned, or even emulated in software. The VMM is much smaller than a traditional OS; the isolation portion of a VMM is perhaps only 10,000 lines of code.

In general, the cost of processor virtualization depends on the workload. User-level processor-bound programs, such as SPEC CPU2006, have zero virtualization overhead because the OS is rarely invoked so everything runs at native speeds. Conversely, I/O-intensive workloads generally are also OS-intensive and execute many system calls (which doing I/O requires) and privileged instructions that can result in high virtualization overhead. The overhead is determined by the number of instructions that must be emulated by the VMM and how slowly they are emulated. Hence, when the guest VMs run the same ISA as the host, as we assume here, the goal of the architecture and the VMM is to run almost all instructions directly on the native hardware. On the other hand, if the I/O-intensive workload is also *I/O-bound*, the cost of processor virtualization can be completely hidden by low processor utilization since it is often waiting for I/O.

Although our interest here is in VMs for improving protection, VMs provide two other benefits that are commercially significant:

1. *Managing software*—VMs provide an abstraction that can run the complete software stack, even including old operating systems such as DOS. A typical deployment might be some VMs running legacy OSes, many running the current stable OS release, and a few testing the next OS release.
2. *Managing hardware*—One reason for multiple servers is to have each application running with its own compatible version of the operating system on separate computers, as this separation can improve dependability. VMs allow these separate software stacks to run independently yet share hardware, thereby consolidating the number of servers. Another example is that some VMMs support migration of a running VM to a different computer, either to balance load or to evacuate from failing hardware.

These two reasons are why cloud-based servers, such as Amazon’s, rely on virtual machines.

Requirements of a Virtual Machine Monitor

What must a VM monitor do? It presents a software interface to guest software, it must isolate the state of guests from each other, and it must protect itself from guest software (including guest OSes). The qualitative requirements are

- Guest software should behave on a VM exactly as if it were running on the native hardware, except for performance-related behavior or limitations of fixed resources shared by multiple VMs.
- Guest software should not be able to change allocation of real system resources directly.

To “virtualize” the processor, the VMM must control just about everything—access to privileged state, address translation, I/O, exceptions and interrupts—even though the guest VM and OS currently running are temporarily using them.

For example, in the case of a timer interrupt, the VMM would suspend the currently running guest VM, save its state, handle the interrupt, determine which guest VM to run next, and then load its state. Guest VMs that rely on a timer interrupt are provided with a virtual timer and an emulated timer interrupt by the VMM.

To be in charge, the VMM must be at a higher privilege level than the guest VM, which generally runs in user mode; this also ensures that the execution of any privileged instruction will be handled by the VMM. The basic requirements of system virtual machines are almost identical to those for paged virtual memory listed above:

- At least two processor modes, system and user.
- A privileged subset of instructions that is available only in system mode, resulting in a trap if executed in user mode. All system resources must be controllable only via these instructions.

(Lack of) Instruction Set Architecture Support for Virtual Machines

If VMs are planned for during the design of the ISA, it’s relatively easy to both reduce the number of instructions that must be executed by a VMM and how long it takes to emulate them. An architecture that allows the VM to execute directly on the hardware earns the title *virtualizable*, and the IBM 370 architecture proudly bears that label.

Alas, since VMs have been considered for desktop and PC-based server applications only fairly recently, most instruction sets were created without virtualization in mind. These culprits include 80x86 and most RISC architectures.

Because the VMM must ensure that the guest system only interacts with virtual resources, a conventional guest OS runs as a user mode program on top of the VMM. Then, if a guest OS attempts to access or modify information related to hardware resources via a privileged instruction—for example, reading or writing the page table pointer—it will trap to the VMM. The VMM can then effect the appropriate changes to corresponding real resources.

Hence, if any instruction that tries to read or write such sensitive information traps when executed in user mode, the VMM can intercept it and support a virtual version of the sensitive information as the guest OS expects.