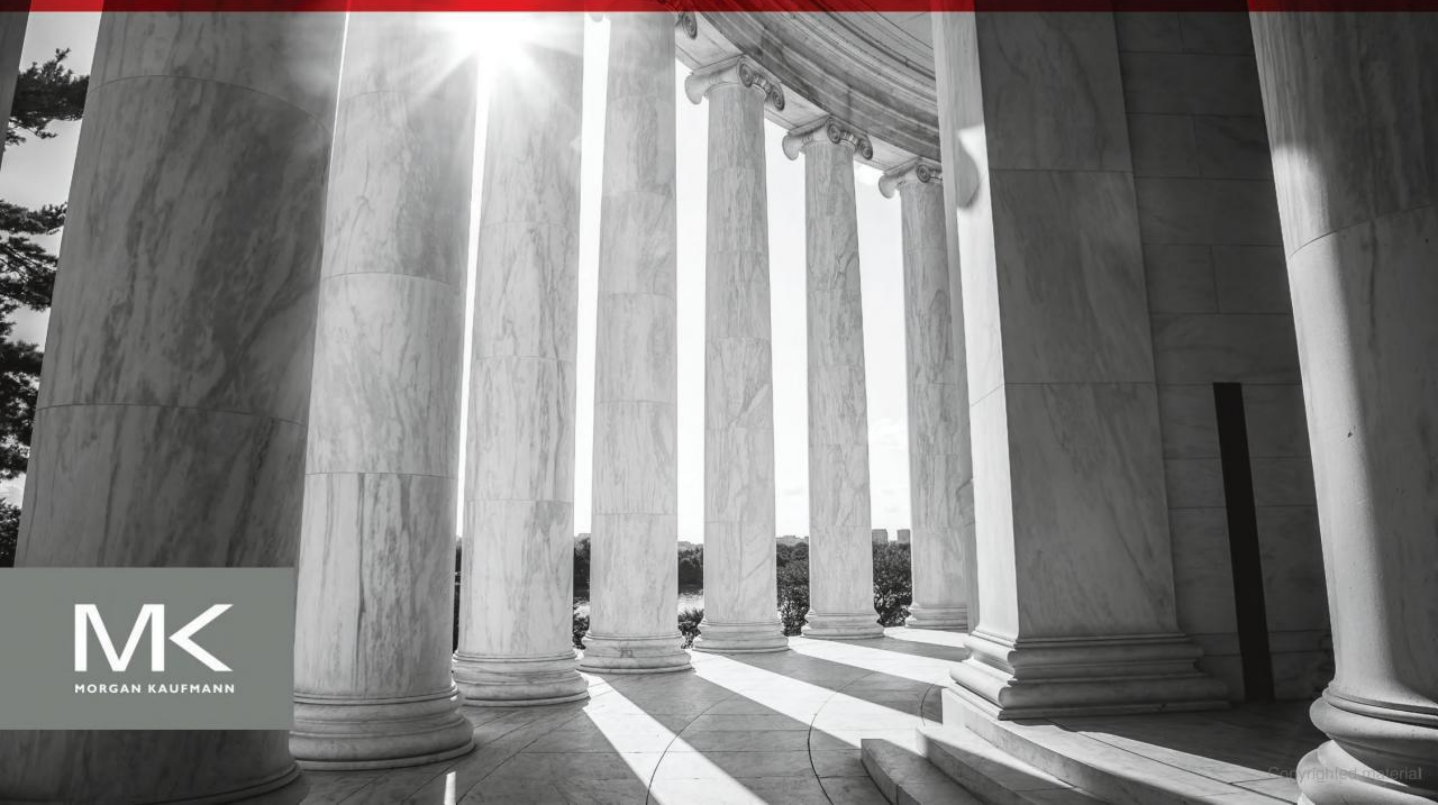
Sixth Edition

John L. Hennessy | David A. Patterson

COMPUTER ARCHITECTURE

A Quantitative Approach



MORGAN KAUFMANN

Computer Architecture
A Quantitative Approach

Sixth 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.
Abhishek Bhattacharjee
Rutgers University
Thomas M. Conte
Georgia Tech
José Duato
Proemisa
Diana Franklin
University of Chicago
David Goldberg
eBay
Norman P. Jouppi
Google
Sheng Li
Intel Labs
Naveen Muralimanohar
HP Labs
Gregory D. Peterson
University of Tennessee
Timothy M. Pinkston
University of Southern California
Parthasarathy Ranganathan
Google
David A. Wood
University of Wisconsin–Madison
Cliff Young
Google
Amr Zaky
University of Santa Clara

MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER

Morgan Kaufmann is an imprint of Elsevier
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States

Notices
Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments 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
A catalog record for this book is available from the Library of Congress

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-0-12-811905-1

Publisher: Katey Birtcher
Acquisition Editor: Stephen Merken
Developmental Editor: Nate McFadden
Production Project Manager: Stalin Viswanathan
Cover Designer: Christian J. Bilbow

Typeset by SPi Global, India

Contents

This page intentionally left blank

Preface

Why We Wrote This Book

Through six editions of this book, our goal has been to describe the basic principles underlying what will be tomorrow's technological developments. Our excitement about the opportunities in computer architecture has not abated, and we echo what we said about the field in the first edition: "It is not a dreary science of paper machines that will never work. No! It's a discipline of keen intellectual interest, requiring the balance of marketplace forces to cost-performance-power, leading to glorious failures and some notable successes."

Our primary objective in writing our first book was to change the way people learn and think about computer architecture. We feel this goal is still valid and important. The field is changing daily and must be studied with real examples and measurements on real computers, rather than simply as a collection of definitions and designs that will never need to be realized. We offer an enthusiastic welcome to anyone who came along with us in the past, as well as to those who are joining us now. Either way, we can promise the same quantitative approach to, and analysis of, real systems.

As with earlier versions, we have strived to produce a new edition that will continue to be as relevant for professional engineers and architects as it is for those involved in advanced computer architecture and design courses. Like the first edition, this edition has a sharp focus on new platforms—personal mobile devices and warehouse-scale computers—and new architectures—specifically, domain-specific architectures. As much as its predecessors, this edition aims to demystify computer architecture through an emphasis on cost-performance-energy trade-offs and good engineering design. We believe that the field has continued to mature and move toward the rigorous quantitative foundation of long-established scientific and engineering disciplines.

This Edition

The ending of Moore's Law and Dennard scaling is having as profound effect on computer architecture as did the switch to multicore. We retain the focus on the extremes in size of computing, with personal mobile devices (PMDs) such as cell phones and tablets as the clients and warehouse-scale computers offering cloud computing as the server. We also maintain the other theme of parallelism in all its forms: *data-level parallelism (DLP)* in Chapters 1 and 4, *instruction-level parallelism (ILP)* in Chapter 3, *thread-level parallelism* in Chapter 5, and *request-level parallelism* (RLP) in Chapter 6.

The most pervasive change in this edition is switching from MIPS to the RISC-V instruction set. We suspect this modern, modular, open instruction set may become a significant force in the information technology industry. It may become as important in computer architecture as Linux is for operating systems.

The newcomer in this edition is Chapter 7, which introduces domain-specific architectures with several concrete examples from industry.

As before, the first three appendices in the book give basics on the RISC-V instruction set, memory hierarchy, and pipelining for readers who have not read a book like *Computer Organization and Design*. To keep costs down but still supply supplemental material that is of interest to some readers, available online at https://www.elsevier.com/books-and-journals/book-companion/9780128119051 are nine more appendices. There are more pages in these appendices than there are in this book!

This edition continues the tradition of using real-world examples to demonstrate the ideas, and the "Putting It All Together" sections are brand new. The "Putting It All Together" sections of this edition include the pipeline organizations and memory hierarchies of the ARM Cortex A8 processor, the Intel core i7 processor, the NVIDIA GTX-280 and GTX-480 GPUs, and one of the Google warehouse-scale computers.

Topic Selection and Organization

As before, we have taken a conservative approach to topic selection, for there are many more interesting ideas in the field than can reasonably be covered in a treatment of basic principles. We have steered away from a comprehensive survey of every architecture a reader might encounter. Instead, our presentation focuses on core concepts likely to be found in any new machine. The key criterion remains that of selecting ideas that have been examined and utilized successfully enough to permit their discussion in quantitative terms.

Our intent has always been to focus on material that is not available in equivalent form from other sources, so we continue to emphasize advanced content wherever possible. Indeed, there are several systems here whose descriptions cannot be found in the literature. (Readers interested strictly in a more basic introduction to computer architecture should read *Computer Organization and Design: The Hardware/Software Interface*.)

An Overview of the Content

Chapter 1 includes formulas for energy, static power, dynamic power, integrated circuit costs, reliability, and availability. (These formulas are also found on the front inside cover.) Our hope is that these topics can be used through the rest of the book. In addition to the classic quantitative principles of computer design and performance measurement, it shows the slowing of performance improvement of general-purpose microprocessors, which is one inspiration for domain-specific architectures.

Our view is that the instruction set architecture is playing less of a role today than in 1990, so we moved this material to Appendix A. It now uses the RISC-V architecture. (For quick review, a summary of the RISC-V ISA can be found on the back inside cover.) For fans of ISAs, Appendix K was revised for this edition and covers 8 RISC architectures (5 for desktop and server use and 3 for embedded use), the 80×86, the DEC VAX, and the IBM 360/370.

We then move onto memory hierarchy in Chapter 2, since it is easy to apply the cost-performance-energy principles to this material, and memory is a critical resource for the rest of the chapters. As in the past edition, Appendix B contains an introductory review of cache principles, which is available in case you need it. Chapter 2 discusses 10 advanced optimizations of caches. The chapter includes virtual machines, which offer advantages in protection, software management, and hardware management, and play an important role in cloud computing. In addition to covering SRAM and DRAM technologies, the chapter includes new material both on Flash memory and on the use of stacked die packaging for extending the memory hierarchy. The PIAT examples are the ARM Cortex A8, which is used in PMDs, and the Intel Core i7, which is used in servers.

Chapter 3 covers the exploitation of instruction-level parallelism in high-performance processors, including superscalar execution, branch prediction (including the new tagged hybrid predictors), speculation, dynamic scheduling, and simultaneous multithreading. As mentioned earlier, Appendix C is a review of pipelining in case you need it. Chapter 3 also surveys the limits of ILP. Like Chapter 2, the PIAT examples are again the ARM Cortex A8 and the Intel Core i7. While the third edition contained a great deal on Itanium and VLIW, this material is now in Appendix H, indicating our view that this architecture did not live up to the earlier claims.

The increasing importance of multimedia applications such as games and video processing has also increased the importance of architectures that can exploit data level parallelism. In particular, there is a rising interest in computing using graphical processing units (GPUs), yet few architects understand how GPUs really work. We decided to write a new chapter in large part to unveil this new style of computer architecture. Chapter 4 starts with an introduction to vector architectures, which acts as a foundation on which to build explanations of multimedia SIMD instruction set extensions and GPUs. (Appendix G goes into even more depth on vector architectures.) This chapter introduces the Roofline performance model and then uses it to compare the Intel Core i7 and the NVIDIA GTX 280 and GTX 480 GPUs. The chapter also describes the Tegra 2 GPU for PMDs.

Chapter 5 describes multicore processors. It explores symmetric and distributed-memory architectures, examining both organizational principles and performance. The primary additions to this chapter include more comparison of multicore organizations, including the organization of multicore-multilevel caches, multicore coherence schemes, and on-chip multicore interconnect. Topics in synchronization and memory consistency models are next. The example is the Intel Core i7. Readers interested in more depth on interconnection networks should read Appendix F, and those interested in larger scale multiprocessors and scientific applications should read Appendix I.

Chapter 6 describes warehouse-scale computers (WSCs). It was extensively revised based on help from engineers at Google and Amazon Web Services. This chapter integrates details on design, cost, and performance of WSCs that few architects are aware of. It starts with the popular MapReduce programming model before describing the architecture and physical implementation of WSCs, including cost. The costs allow us to explain the emergence of cloud computing, whereby it can be cheaper to compute using WSCs in the cloud than in your local datacenter. The PIAT example is a description of a Google WSC that includes information published for the first time in this book.

The new Chapter 7 motivates the need for Domain-Specific Architectures (DSAs). It draws guiding principles for DSAs based on the four examples of DSAs. Each DSA corresponds to chips that have been deployed in commercial settings. We also explain why we expect a renaissance in computer architecture via DSAs given that single-thread performance of general-purpose microprocessors has stalled.

This brings us to Appendices A through M. Appendix A covers principles of ISAs, including RISC-V, and Appendix K describes 64-bit versions of RISC V, ARM, MIPS, Power, and SPARC and their multimedia extensions. It also includes some classic architectures (80x86, VAX, and IBM 360/370) and popular embedded instruction sets (Thumb-2, microMIPS, and RISC V C). Appendix H is related, in that it covers architectures and compilers for VLIW ISAs.

As mentioned earlier, Appendix B and Appendix C are tutorials on basic caching and pipelining concepts. Readers relatively new to caching should read Appendix B before Chapter 2, and those new to pipelining should read Appendix C before Chapter 3.

Appendix D, “Storage Systems,” has an expanded discussion of reliability and availability, a tutorial on RAID with a description of RAID 6 schemes, and rarely found failure statistics of real systems. It continues to provide an introduction to queuing theory and I/O performance benchmarks. We evaluate the cost, performance, and reliability of a real cluster: the Internet Archive. The “Putting It All Together” example is the NetApp FAS6000 filer.

Appendix E, by Thomas M. Conte, consolidates the embedded material in one place.

Appendix F, on interconnection networks, is revised by Timothy M. Pinkston and José Duato. Appendix G, written originally by Krste Asanović, includes a description of vector processors. We think these two appendices are some of the best material we know of on each topic.

Appendix H describes VLIW and EPIC, the architecture of Itanium.

Appendix I describes parallel processing applications and coherence protocols for larger-scale, shared-memory multiprocessing. Appendix J, by David Goldberg, describes computer arithmetic.

Appendix L, by Abhishek Bhattacharjee, is new and discusses advanced techniques for memory management, focusing on support for virtual machines and design of address translation for very large address spaces. With the growth in clouds processors, these architectural enhancements are becoming more important.

Appendix M collects the "Historical Perspective and References" from each chapter into a single appendix. It attempts to give proper credit for the ideas in each chapter and a sense of the history surrounding the inventions. We like to think of this as presenting the human drama of computer design. It also supplies references that the student of architecture may want to pursue. If you have time, we recommend reading some of the classic papers in the field that are mentioned in these sections. It is both enjoyable and educational to hear the ideas directly from the creators. "Historical Perspective" was one of the most popular sections of prior editions.

Navigating the Text

There is no single best order in which to approach these chapters and appendices, except that all readers should start with Chapter 1. If you don't want to read everything, here are some suggested sequences:

- *Memory Hierarchy:* Appendix B, Chapter 2, and Appendices D and M.
- *Instruction-Level Parallelism:* Appendix C, Chapter 3, and Appendix H
- *Data-Level Parallelism:* Chapters 4, 6, and 7, Appendix G
- *Thread-Level Parallelism:* Chapter 5, Appendices F and I
- *Request-Level Parallelism:* Chapter 6
- *ISA:* Appendices A and K

Appendix E can be read at any time, but it might work best if read after the ISA and cache sequences. Appendix J can be read whenever arithmetic moves you. You should read the corresponding portion of Appendix M after you complete each chapter.

Chapter Structure

The material we have selected has been stretched upon a consistent framework that is followed in each chapter. We start by explaining the ideas of a chapter. These ideas are followed by a "Crosscutting Issues" section, a feature that shows how the ideas covered in one chapter interact with those given in other chapters. This is

followed by a "Putting It All Together" section that ties these ideas together by showing how they are used in a real machine.

Next in the sequence is "Fallacies and Pitfalls," which lets readers learn from the mistakes of others. We show examples of common misunderstandings and architectural traps that are difficult to avoid even when you know they are lying in wait for you. The "Fallacies and Pitfalls" sections is one of the most popular sections of the book. Each chapter ends with a "Concluding Remarks" section.

Case Studies With Exercises

Each chapter ends with case studies and accompanying exercises. Authored by experts in industry and academia, the case studies explore key chapter concepts and verify understanding through increasingly challenging exercises. Instructors should find the case studies sufficiently detailed and robust to allow them to create their own additional exercises.

Brackets for each exercise (<chapter.section>) indicate the text sections of primary relevance to completing the exercise. We hope this helps readers to avoid exercises for which they haven't read the corresponding section, in addition to providing the source for review. Exercises are rated, to give the reader a sense of the amount of time required to complete an exercise:

[10] Less than 5 min (to read and understand)

[15] 5–15 min for a full answer

[20] 15–20 min for a full answer

[25] 1 h for a full written answer

[30] Short programming project: less than 1 full day of programming

[40] Significant programming project: 2 weeks of elapsed time

[Discussion] Topic for discussion with others

Solutions to the case studies and exercises are available for instructors who register at *textbooks.elsevier.com*.

Supplemental Materials

A variety of resources are available online at https://www.elsevier.com/books/computer-architecture/hennessy/978-0-12-811905-1, including the following:

- Reference appendices, some guest authored by subject experts, covering a range of advanced topics
- Historical perspectives material that explores the development of the key ideas presented in each of the chapters in the text

- Instructor slides in PowerPoint
- Figures from the book in PDF, EPS, and PPT formats
- Links to related material on the Web
- List of errata

New materials and links to other resources available on the Web will be added on a regular basis.

Helping Improve This Book

Finally, it is possible to make money while reading this book. (Talk about cost performance!) If you read the Acknowledgments that follow, you will see that we went to great lengths to correct mistakes. Since a book goes through many printings, we have the opportunity to make even more corrections. If you uncover any remaining resilient bugs, please contact the publisher by electronic mail (ca6bugs@mkp.com).

We welcome general comments to the text and invite you to send them to a separate email address at ca6comments@mkp.com.

Concluding Remarks

Once again, this book is a true co-authorship, with each of us writing half the chapters and an equal share of the appendices. We can't imagine how long it would have taken without someone else doing half the work, offering inspiration when the task seemed hopeless, providing the key insight to explain a difficult concept, supplying over-the-weekend reviews of chapters, and commiserating when the weight of our other obligations made it hard to pick up the pen.

Thus, once again, we share equally the blame for what you are about to read.

John Hennessy ■ *David Patterson*

This page intentionally left blank

Acknowledgments

Although this is only the sixth edition of this book, we have actually created ten different versions of the text: three versions of the first edition (alpha, beta, and final) and two versions of the second, third, and fourth editions (beta and final). Along the way, we have received help from hundreds of reviewers and users. Each of these people has helped make this book better. Thus, we have chosen to list all of the people who have made contributions to some version of this book.

Contributors to the Sixth Edition

Like prior editions, this is a community effort that involves scores of volunteers. Without their help, this edition would not be nearly as polished.

Reviewers

Jason D. Bakos, University of South Carolina; Rajeev Balasubramonian, University of Utah; Jose Delgado-Frias, Washington State University; Diana Franklin, The University of Chicago; Norman P. Jouppi, Google; Hugh C. Lauer, Worcester Polytechnic Institute; Gregory Peterson, University of Tennessee; Bill Pierce, Hood College; Parthasarathy Ranganathan, Google; William H. Robinson, Vanderbilt University; Pat Stakem, Johns Hopkins University; Cliff Young, Google; Amr Zaky, University of Santa Clara; Gerald Zarnett, Ryerson University; Huiyang Zhou, North Carolina State University.

Members of the University of California-Berkeley Par Lab and RAD Lab who gave frequent reviews of Chapters 1, 4, and 6 and shaped the explanation of GPUs and WSCs: Krste Asanović, Michael Armbrust, Scott Beamer, Sarah Bird, Bryan Catanzaro, Jike Chong, Henry Cook, Derrick Coetzee, Randy Katz, Yunsup Lee, Leo Meyervich, Mark Murphy, Zhangxi Tan, Vasily Volkov, and Andrew Waterman.

Appendices

Krste Asanović, University of California, Berkeley (Appendix G); Abhishek Bhattacharjee, Rutgers University (Appendix L); Thomas M. Conte, North Carolina State University (Appendix E); José Duato, Universitat Politècnica de

Valèncía and Simula (Appendix F); David Goldberg, Xerox PARC (Appendix J); Timothy M. Pinkston, University of Southern California (Appendix F).

José Flich of the Universidad Politécnica de Valencia provided significant contributions to the updating of Appendix F.

Case Studies With Exercises

Jason D. Bakos, University of South Carolina (Chapters 3 and 4); Rajeev Balasubramonian, University of Utah (Chapter 2); Diana Franklin, The University of Chicago (Chapter 1 and Appendix C); Norman P. Jouppi, Google, (Chapter 2); Naveen Muralimanohar, HP Labs (Chapter 2); Gregory Peterson, University of Tennessee (Appendix A); Parthasarathy Ranganathan, Google (Chapter 6); Cliff Young, Google (Chapter 7); Amr Zaky, University of Santa Clara (Chapter 5 and Appendix B).

Jichuan Chang, Junwhan Ahn, Rama Govindaraju, and Milad Hashemi assisted in the development and testing of the case studies and exercises for Chapter 6.

Additional Material

John Nickolls, Steve Keckler, and Michael Toksvig of NVIDIA (Chapter 4 NVIDIA GPUs); Victor Lee, Intel (Chapter 4 comparison of Core i7 and GPU); John Shalf, LBNL (Chapter 4 recent vector architectures); Sam Williams, LBNL (Roofline model for computers in Chapter 4); Steve Blackburn of Australian National University and Kathryn McKinley of University of Texas at Austin (Intel performance and power measurements in Chapter 5); Luiz Barroso, Urs Hölzle, Jimmy Clidaris, Bob Felderman, and Chris Johnson of Google (the Google WSC in Chapter 6); James Hamilton of Amazon Web Services (power distribution and cost model in Chapter 6).

Jason D. Bakos of the University of South Carolina updated the lecture slides for this edition.

This book could not have been published without a publisher, of course. We wish to thank all the Morgan Kaufmann/Elsevier staff for their efforts and support. For this fifth edition, we particularly want to thank our editors Nate McFadden and Steve Merken, who coordinated surveys, development of the case studies and exercises, manuscript reviews, and the updating of the appendices.

We must also thank our university staff, Margaret Rowland and Roxana Infante, for countless express mailings, as well as for holding down the fort at Stanford and Berkeley while we worked on the book.

Our final thanks go to our wives for their suffering through increasingly early mornings of reading, thinking, and writing.

Contributors to Previous Editions

Reviewers

George Adams, Purdue University; Sarita Adve, University of Illinois at Urbana-Champaign; Jim Archibald, Brigham Young University; Krste Asanović, Massachusetts Institute of Technology; Jean-Loup Baer, University of Washington; Paul Barr, Northeastern University; Rajendra V. Boppana, University of Texas, San Antonio; Mark Brehob, University of Michigan; Doug Burger, University of Texas, Austin; John Burger, SGI; Michael Butler; Thomas Casavant; Rohit Chandra; Peter Chen, University of Michigan; the classes at SUNY Stony Brook, Carnegie Mellon, Stanford, Clemson, and Wisconsin; Tim Coe, Vitesse Semiconductor; Robert P. Colwell; David Cummings; Bill Dally; David Douglas; José Duato, Universitat Politècnica de València and Simula; Anthony Duben, Southeast Missouri State University; Susan Eggers, University of Washington; Joel Emer; Barry Fagin, Dartmouth; Joel Ferguson, University of California, Santa Cruz; Carl Feynman; David Filo; Josh Fisher, Hewlett-Packard Laboratories; Rob Fowler, DIKU; Mark Franklin, Washington University (St. Louis); Kourosh Gharachorloo; Nikolas Gloy, Harvard University; David Goldberg, Xerox Palo Alto Research Center; Antonio González, Intel and Universitat Politècnica de Catalunya; James Goodman, University of Wisconsin-Madison; Sudhanva Gurumurthi, University of Virginia; David Harris, Harvey Mudd College; John Heinlein; Mark Heinrich, Stanford; Daniel Helman, University of California, Santa Cruz; Mark D. Hill, University of Wisconsin-Madison; Martin Hopkins, IBM; Jerry Huck, Hewlett-Packard Laboratories; Wen-mei Hwu, University of Illinois at Urbana-Champaign; Mary Jane Irwin, Pennsylvania State University; Truman Joe; Norm Jouppi; David Kaeli, Northeastern University; Roger Kieckhafer, University of Nebraska; Lev G. Kirischian, Ryerson University; Earl Killian; Allan Knies, Purdue University; Don Knuth; Jeff Kuskin, Stanford; James R. Larus, Microsoft Research; Corinna Lee, University of Toronto; Hank Levy; Kai Li, Princeton University; Lori Liebrock, University of Alaska, Fairbanks; Mikko Lipasti, University of Wisconsin-Madison; Gyula A. Mago, University of North Carolina, Chapel Hill; Bryan Martin; Norman Matloff; David Meyer; William Michalson, Worcester Polytechnic Institute; James Mooney; Trevor Mudge, University of Michigan; Ramadass Nagarajan, University of Texas at Austin; David Nagle, Carnegie Mellon University; Todd Narter; Victor Nelson; Vojin Oklobdzija, University of California, Berkeley; Kunle Olukotun, Stanford University; Bob Owens, Pennsylvania State University; Greg Papadapoulous, Sun Microsystems; Joseph Pfeiffer; Keshav Pingali, Cornell University; Timothy M. Pinkston, University of Southern California; Bruno Preiss, University of Waterloo; Steven Przybylski; Jim Quinlan; Andras Radics; Kishore Ramachandran, Georgia Institute of Technology; Joseph Rameh, University of Texas, Austin; Anthony Reeves, Cornell University; Richard Reid, Michigan State University; Steve Reinhardt, University of Michigan; David Rennels, University of California, Los Angeles; Arnold L. Rosenberg, University of Massachusetts, Amherst; Kaushik Roy, Purdue

University; Emilio Salgueiro, Unysis; Karthikeyan Sankaralingam, University of Texas at Austin; Peter Schnorf; Margo Seltzer; Behrooz Shirazi, Southern Methodist University; Daniel Siewiorek, Carnegie Mellon University; J. P. Singh, Princeton; Ashok Singhal; Jim Smith, University of Wisconsin-Madison; Mike Smith, Harvard University; Mark Smotherman, Clemson University; Gurindar Sohi, University of Wisconsin-Madison; Arun Somani, University of Washington; Gene Tagliarin, Clemson University; Shyamkumar Thoziyoor, University of Notre Dame; Evan Tick, University of Oregon; Akhilesh Tyagi, University of North Carolina, Chapel Hill; Dan Upton, University of Virginia; Mateo Valero, Universidad Politécnica de Cataluña, Barcelona; Anujan Varma, University of California, Santa Cruz; Thorsten von Eicken, Cornell University; Hank Walker, Texas A&M; Roy Want, Xerox Palo Alto Research Center; David Weaver, Sun Microsystems; Shlomo Weiss, Tel Aviv University; David Wells; Mike Westall, Clemson University; Maurice Wilkes; Eric Williams; Thomas Willis, Purdue University; Malcolm Wing; Larry Wittie, SUNY Stony Brook; Ellen Witte Zegura, Georgia Institute of Technology; Sotirios G. Ziavras, New Jersey Institute of Technology.

Appendices

The vector appendix was revised by Krste Asanović of the Massachusetts Institute of Technology. The floating-point appendix was written originally by David Goldberg of Xerox PARC.

Exercises

George Adams, Purdue University; Todd M. Bezenek, University of Wisconsin-Madison (in remembrance of his grandmother Ethel Eshom); Susan Eggers; Anoop Gupta; David Hayes; Mark Hill; Allan Knies; Ethan L. Miller, University of California, Santa Cruz; Parthasarathy Ranganathan, Compaq Western Research Laboratory; Brandon Schwartz, University of Wisconsin-Madison; Michael Scott; Dan Siewiorek; Mike Smith; Mark Smotherman; Evan Tick; Thomas Willis.

Case Studies With Exercises

Andrea C. Arpaci-Dusseau, University of Wisconsin-Madison; Remzi H. Arpaci-Dusseau, University of Wisconsin-Madison; Robert P. Colwell, R&E Colwell & Assoc., Inc.; Diana Franklin, California Polytechnic State University, San Luis Obispo; Wen-mei W. Hwu, University of Illinois at Urbana-Champaign; Norman P. Jouppi, HP Labs; John W. Sias, University of Illinois at Urbana-Champaign; David A. Wood, University of Wisconsin-Madison.

Special Thanks

Duane Adams, Defense Advanced Research Projects Agency; Tom Adams; Sarita Adve, University of Illinois at Urbana-Champaign; Anant Agarwal; Dave

Albonesi, University of Rochester; Mitch Alsup; Howard Alt; Dave Anderson; Peter Ashenden; David Bailey; Bill Bandy, Defense Advanced Research Projects Agency; Luiz Barroso, Compaq's Western Research Lab; Andy Bechtolsheim; *C. Gordon* Bell; Fred Berkowitz; John Best, IBM; Dileep Bhandarkar; Jeff Bier, BDTI; Mark Birman; David Black; David Boggs; Jim Brady; Forrest Brewer; Aaron Brown, University of California, Berkeley; E. Bugnion, Compaq's Western Research Lab; Alper Buyuktosunoglu, University of Rochester; Mark Callaghan; Jason F. Cantin; Paul Carrick; Chen-Chung Chang; Lei Chen, University of Rochester; Pete Chen; Nhan Chu; Doug Clark, Princeton University; Bob Cmelik; John Crawford; Zarka Cvetanovic; Mike Dahlin, University of Texas, Austin; Merrick Darley; the staff of the DEC Western Research Laboratory; John DeRosa; Lloyd Dickman; J. Ding; Susan Eggers, University of Washington; Wael El-Essawy, University of Rochester; Patty Enriquez, Mills; Milos Ercegovac; Robert Garner; K. Gharachorloo, Compaq's Western Research Lab; Garth Gibson; Ronald Greenberg; Ben Hao; John Henning, Compaq; Mark Hill, University of Wisconsin-Madison; Danny Hillis; David Hodges; Urs Hölzle, Google; David Hough; Ed Hudson; Chris Hughes, University of Illinois at Urbana-Champaign; Mark Johnson; Lewis Jordan; Norm Jouppi; William Kahan; Randy Katz; Ed Kelly; Richard Kessler; Les Kohn; John Kowaleski, Compaq Computer Corp; Dan Lambright; Gary Lauterbach, Sun Microsystems; Corinna Lee; Ruby Lee; Don Lewine; Chao-Huang Lin; Paul Losleben, Defense Advanced Research Projects Agency; Yung-Hsiang Lu; Bob Lucas, Defense Advanced Research Projects Agency; Ken Lutz; Alan Mainwaring, Intel Berkeley Research Labs; Al Marston; Rich Martin, Rutgers; John Mashey; Luke McDowell; Sebastian Mirolo, Trimedia Corporation; Ravi Murthy; Biswadeep Nag; Lisa Noordergraaf, Sun Microsystems; Bob Parker, Defense Advanced Research Projects Agency; Vern Paxson, Center for Internet Research; Lawrence Prince; Steven Przybylski; Mark Pullen, Defense Advanced Research Projects Agency; Chris Rowen; Margaret Rowland; Greg Semeraro, University of Rochester; Bill Shannon; Behrooz Shirazi; Robert Shomler; Jim Slager; Mark Smotherman, Clemson University; the SMT research group at the University of Washington; Steve Squires, Defense Advanced Research Projects Agency; Ajay Sreekanth; Darren Staples; Charles Stapper; Jorge Stolfi; Peter Stoll; the students at Stanford and Berkeley who endured our first attempts at creating this book; Bob Supnik; Steve Swanson; Paul Taysom; Shreekant Thakkar; Alexander Thomasian, New Jersey Institute of Technology; John Toole, Defense Advanced Research Projects Agency; Kees A. Vissers, Trimedia Corporation; Willa Walker; David Weaver; Ric Wheeler, EMC; Maurice Wilkes; Richard Zimmerman.

John Hennessy ■ *David Patterson*

1

Fundamentals of Quantitative Design and Analysis

An iPod, a phone, an Internet mobile communicator… these are NOT three separate devices! And we are calling it iPhone! Today Apple is going to reinvent the phone. And here it is.

Steve Jobs, January 9, 2007

New information and communications technologies, in particular high-speed Internet, are changing the way companies do business, transforming public service delivery and democratizing innovation. With 10 percent increase in high speed Internet connections, economic growth increases by 1.3 percent.

The World Bank, July 28, 2009

Computer Architecture. https://doi.org/10.1016/B978-0-12-811905-1.00001-8

1.1 Introduction

Computer technology has made incredible progress in the roughly 70 years since the first general-purpose electronic computer was created. Today, less than $500 will purchase a cell phone that has as much performance as the world's fastest computer bought in 1993 for $50 million. This rapid improvement has come both from advances in the technology used to build computers and from innovations in computer design.

Although technological improvements historically have been fairly steady, progress arising from better computer architectures has been much less consistent. During the first 25 years of electronic computers, both forces made a major contribution, delivering performance improvement of about 25% per year. The late 1970s saw the emergence of the microprocessor. The ability of the microprocessor to ride the improvements in integrated circuit technology led to a higher rate of performance improvement—roughly 35% growth per year.

This growth rate, combined with the cost advantages of a mass-produced microprocessor, led to an increasing fraction of the computer business being based on microprocessors. In addition, two significant changes in the computer marketplace made it easier than ever before to succeed commercially with a new architecture. First, the virtual elimination of assembly language programming reduced the need for object-code compatibility. Second, the creation of standardized, vendor-independent operating systems, such as UNIX and its clone, Linux, lowered the cost and risk of bringing out a new architecture.

These changes made it possible to develop successfully a new set of architectures with simpler instructions, called RISC (Reduced Instruction Set Computer) architectures, in the early 1980s. The RISC-based machines focused the attention of designers on two critical performance techniques, the exploitation of *instruction-level parallelism* (initially through pipelining and later through multiple instruction issue) and the use of caches (initially in simple forms and later using more sophisticated organizations and optimizations).

The RISC-based computers raised the performance bar, forcing prior architectures to keep up or disappear. The Digital Equipment Vax could not, and so it was replaced by a RISC architecture. Intel rose to the challenge, primarily by translating 80x86 instructions into RISC-like instructions internally, allowing it to adopt many of the innovations first pioneered in the RISC designs. As transistor counts soared in the late 1990s, the hardware overhead of translating the more complex x86 architecture became negligible. In low-end applications, such as cell phones, the cost in power and silicon area of the x86-translation overhead helped lead to a RISC architecture, ARM, becoming dominant.

Figure 1.1 shows that the combination of architectural and organizational enhancements led to 17 years of sustained growth in performance at an annual rate of over 50%—a rate that is unprecedented in the computer industry.

The effect of this dramatic growth rate during the 20th century was fourfold. First, it has significantly enhanced the capability available to computer users. For many applications, the highest-performance microprocessors outperformed the supercomputer of less than 20 years earlier.

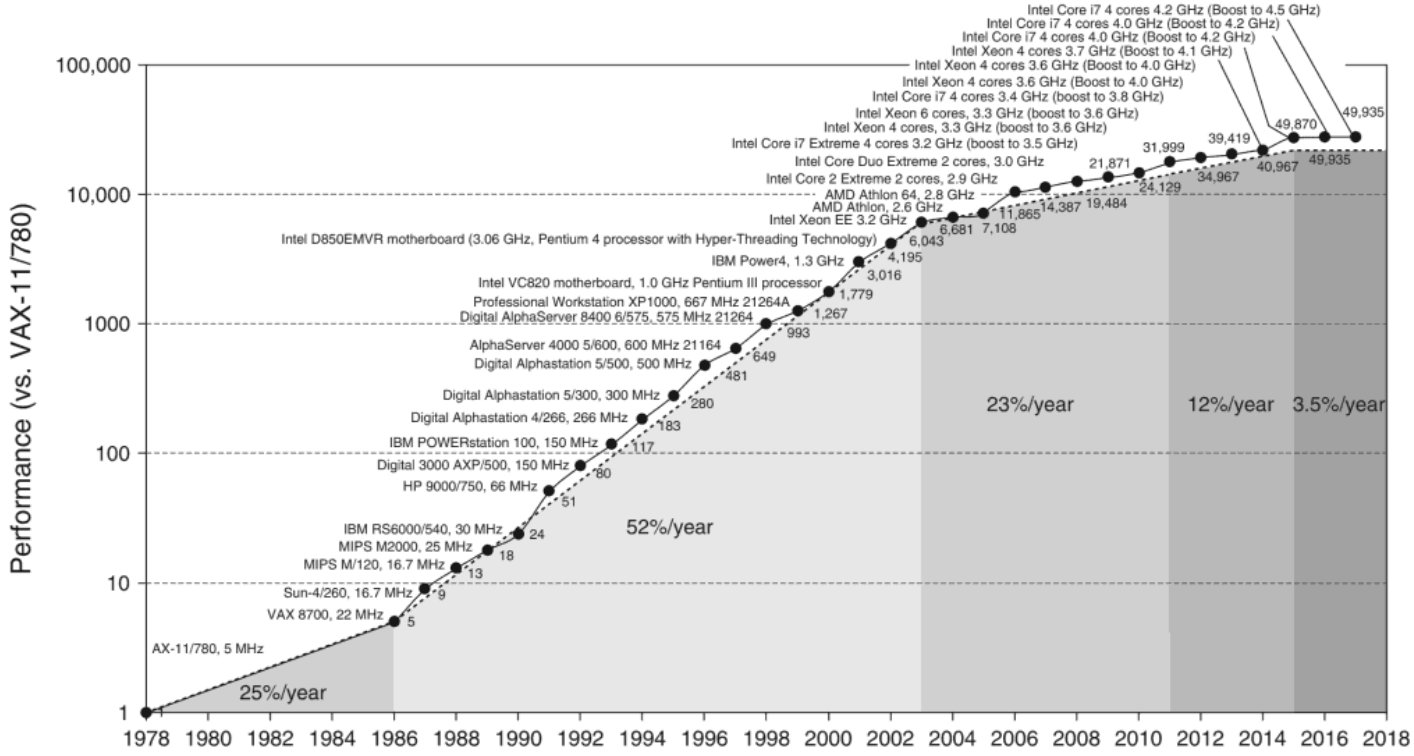


Figure 1.1 Growth in processor performance over 40 years. This chart plots program performance relative to the VAX 11/780 as measured by the SPEC integer benchmarks (see Section 1.8). Prior to the mid-1980s, growth in processor performance was largely technology-driven and averaged about 22% per year, or doubling performance every 3.5 years. The increase in growth to about 52% starting in 1986, or doubling every 2 years, is attributable to more advanced architectural and organizational ideas typified in RISC architectures. By 2003 this growth led to a difference in performance of an approximate factor of 25 versus the performance that would have occurred if it had continued at the 22% rate. In 2003 the limits of power due to the end of Dennard scaling and the available instruction-level parallelism slowed uniprocessor performance to 23% per year until 2011, or doubling every 3.5 years. (The fastest SPECintbase performance since 2007 has had automatic parallelization turned on, so uniprocessor speed is harder to gauge. These results are limited to single-chip systems with usually four cores per chip.) From 2011 to 2015, the annual improvement was less than 12%, or doubling every 8 years in part due to the limits of parallelism of Amdahl's Law. Since 2015, with the end of Moore's Law, improvement has been just 3.5% per year, or doubling every 20 years! Performance for floating-point-oriented calculations follows the same trends, but typically has 1% to 2% higher annual growth in each shaded region. Figure 1.11 on page 27 shows the improvement in clock rates for these same eras. Because SPEC has changed over the years, performance of newer machines is estimated by a scaling factor that relates the performance for different versions of SPEC: SPEC89, SPEC92, SPEC95, SPEC2000, and SPEC2006. There are too few results for SPEC2017 to plot yet.

Second, this dramatic improvement in cost-performance led to new classes of computers. Personal computers and workstations emerged in the 1980s with the availability of the microprocessor. The past decade saw the rise of smart cell phones and tablet computers, which many people are using as their primary computing platforms instead of PCs. These mobile client devices are increasingly using the Internet to access warehouses containing 100,000 servers, which are being designed as if they were a single gigantic computer.

Third, improvement of semiconductor manufacturing as predicted by Moore's law has led to the dominance of microprocessor-based computers across the entire range of computer design. Minicomputers, which were traditionally made from off-the-shelf logic or from gate arrays, were replaced by servers made by using microprocessors. Even mainframe computers and high-performance supercomputers are all collections of microprocessors.

The preceding hardware innovations led to a renaissance in computer design, which emphasized both architectural innovation and efficient use of technology improvements. This rate of growth compounded so that by 2003, high-performance microprocessors were 7.5 times as fast as what would have been obtained by relying solely on technology, including improved circuit design, that is, 52% per year versus 35% per year.

This hardware renaissance led to the fourth impact, which was on software development. This 50,000-fold performance improvement since 1978 (see Figure 1.1) allowed modern programmers to trade performance for productivity. In place of performance-oriented languages like C and C++, much more programming today is done in managed programming languages like Java and Scala. Moreover, scripting languages like JavaScript and Python, which are even more productive, are gaining in popularity along with programming frameworks like AngularJS and Django. To maintain productivity and try to close the performance gap, interpreters with just-in-time compilers and trace-based compiling are replacing the traditional compiler and linker of the past. Software deployment is changing as well, with Software as a Service (SaaS) used over the Internet replacing shrink-wrapped software that must be installed and run on a local computer.

The nature of applications is also changing. Speech, sound, images, and video are becoming increasingly important, along with predictable response time that is so critical to the user experience. An inspiring example is Google Translate. This application lets you hold up your cell phone to point its camera at an object, and the image is sent wirelessly over the Internet to a warehouse-scale computer (WSC) that recognizes the text in the photo and translates it into your native language. You can also speak into it, and it will translate what you said into audio output in another language. It translates text in 90 languages and voice in 15 languages.

Alas, Figure 1.1 also shows that this 17-year hardware renaissance is over. The fundamental reason is that two characteristics of semiconductor processes that were true for decades no longer hold.

In 1974 Robert Dennard observed that power density was constant for a given area of silicon even as you increased the number of transistors because of smaller dimensions of each transistor. Remarkably, transistors could go faster but use less

power. *Dennard scaling* ended around 2004 because current and voltage couldn't keep dropping and still maintain the dependability of integrated circuits.

This change forced the microprocessor industry to use multiple efficient processors or cores instead of a single inefficient processor. Indeed, in 2004 Intel canceled its high-performance uniprocessor projects and joined others in declaring that the road to higher performance would be via multiple processors per chip rather than via faster uniprocessors. This milestone signaled a historic switch from relying solely on instruction-level parallelism (ILP), the primary focus of the first three editions of this book, to *data-level parallelism* (DLP) and *thread-level parallelism* (TLP), which were featured in the fourth edition and expanded in the fifth edition. The fifth edition also added WSCs and *request-level parallelism* (RLP), which is expanded in this edition. Whereas the compiler and hardware conspire to exploit ILP implicitly without the programmer's attention, DLP, TLP, and RLP are explicitly parallel, requiring the restructuring of the application so that it can exploit explicit parallelism. In some instances, this is easy; in many, it is a major new burden for programmers.

Amdahl's Law (Section 1.9) prescribes practical limits to the number of useful cores per chip. If 10% of the task is serial, then the maximum performance benefit from parallelism is 10 no matter how many cores you put on the chip.

The second observation that ended recently is *Moore's Law*. In 1965 Gordon Moore famously predicted that the number of transistors per chip would double every year, which was amended in 1975 to every two years. That prediction lasted for about 50 years, but no longer holds. For example, in the 2010 edition of this book, the most recent Intel microprocessor had 1,170,000,000 transistors. If Moore's Law had continued, we could have expected microprocessors in 2016 to have 18,720,000,000 transistors. Instead, the equivalent Intel microprocessor has just 1,750,000,000 transistors, or off by a factor of 10 from what Moore's Law would have predicted.

The combination of

- transistors no longer getting much better because of the slowing of Moore's Law and the end of Dinnard scaling,
- the unchanging power budgets for microprocessors,
- the replacement of the single power-hungry processor with several energy-efficient processors, and
- the limits to multiprocessing to achieve Amdahl's Law

caused improvements in processor performance to slow down, that is, to *double every 20 years,* rather than every 1.5 years as it did between 1986 and 2003 (see Figure 1.1).

The only path left to improve energy-performance-cost is specialization. Future microprocessors will include several domain-specific cores that perform only one class of computations well, but they do so remarkably better than general-purpose cores. The new Chapter 7 in this edition introduces *domain-specific architectures*.

This text is about the architectural ideas and accompanying compiler improvements that made the incredible growth rate possible over the past century, the reasons for the dramatic change, and the challenges and initial promising approaches to architectural ideas, compilers, and interpreters for the 21st century. At the core is a quantitative approach to computer design and analysis that uses empirical observations of programs, experimentation, and simulation as its tools. It is this style and approach to computer design that is reflected in this text. The purpose of this chapter is to lay the quantitative foundation on which the following chapters and appendices are based.

This book was written not only to explain this design style but also to stimulate you to contribute to this progress. We believe this approach will serve the computers of the future just as it worked for the implicitly parallel computers of the past.

1.2 Classes of Computers

These changes have set the stage for a dramatic change in how we view computing, computing applications, and the computer markets in this new century. Not since the creation of the personal computer have we seen such striking changes in the way computers appear and in how they are used. These changes in computer use have led to five diverse computing markets, each characterized by different applications, requirements, and computing technologies. Figure 1.2 summarizes these mainstream classes of computing environments and their important characteristics.

Internet of Things/Embedded Computers

Embedded computers are found in everyday machines: microwaves, washing machines, most printers, networking switches, and all automobiles. The phrase

Feature	Personal mobile device (PMD)	Desktop	Server	Clusters/warehouse-scale computer	Internet of things/ embedded
Price of system	$100–$1000	$300–$2500	$5000–$10,000,000	$100,000–$200,000,000	$10–$100,000
Price of microprocessor	$10–$100	$50–$500	$200–$2000	$50–$250	$0.01–$100
Critical system design issues	Cost, energy, media performance, responsiveness	Price-performance, energy, graphics performance	Throughput, availability, scalability, energy	Price-performance, throughput, energy proportionality	Price, energy, application-specific performance

Figure 1.2 A summary of the five mainstream computing classes and their system characteristics. Sales in 2015 included about 1.6 billion PMDs (90% cell phones), 275 million desktop PCs, and 15 million servers. The total number of embedded processors sold was nearly 19 billion. In total, 14.8 billion ARM-technology-based chips were shipped in 2015. Note the wide range in system price for servers and embedded systems, which go from USB keys to network routers. For servers, this range arises from the need for very large-scale multiprocessor systems for high-end transaction processing.

Internet of Things (IoT) refers to embedded computers that are connected to the Internet, typically wirelessly. When augmented with sensors and actuators, IoT devices collect useful data and interact with the physical world, leading to a wide variety of "smart" applications, such as smart watches, smart thermostats, smart speakers, smart cars, smart homes, smart grids, and smart cities.

Embedded computers have the widest spread of processing power and cost. They include 8-bit to 32-bit processors that may cost one penny, and high-end 64-bit processors for cars and network switches that cost $100. Although the range of computing power in the embedded computing market is very large, price is a key factor in the design of computers for this space. Performance requirements do exist, of course, but the primary goal often meets the performance need at a minimum price, rather than achieving more performance at a higher price. The projections for the number of IoT devices in 2020 range from 20 to 50 billion.

Most of this book applies to the design, use, and performance of embedded processors, whether they are off-the-shelf microprocessors or microprocessor cores that will be assembled with other special-purpose hardware.

Unfortunately, the data that drive the quantitative design and evaluation of other classes of computers have not yet been extended successfully to embedded computing (see the challenges with EEMBC, for example, in Section 1.8). Hence we are left for now with qualitative descriptions, which do not fit well with the rest of the book. As a result, the embedded material is concentrated in Appendix E. We believe a separate appendix improves the flow of ideas in the text while allowing readers to see how the differing requirements affect embedded computing.

Personal Mobile Device

Personal mobile device (PMD) is the term we apply to a collection of wireless devices with multimedia user interfaces such as cell phones, tablet computers, and so on. Cost is a prime concern given the consumer price for the whole product is a few hundred dollars. Although the emphasis on energy efficiency is frequently driven by the use of batteries, the need to use less expensive packaging—plastic versus ceramic—and the absence of a fan for cooling also limit total power consumption. We examine the issue of energy and power in more detail in Section 1.5. Applications on PMDs are often web-based and media-oriented, like the previously mentioned Google Translate example. Energy and size requirements lead to use of Flash memory for storage (Chapter 2) instead of magnetic disks.

The processors in a PMD are often considered embedded computers, but we are keeping them as a separate category because PMDs are platforms that can run externally developed software, and they share many of the characteristics of desktop computers. Other embedded devices are more limited in hardware and software sophistication. We use the ability to run third-party software as the dividing line between nonembedded and embedded computers.

Responsiveness and predictability are key characteristics for media applications. A *real-time performance* requirement means a segment of the application has an absolute maximum execution time. For example, in playing a video on a

PMD, the time to process each video frame is limited, since the processor must accept and process the next frame shortly. In some applications, a more nuanced requirement exists: the average time for a particular task is constrained as well as the number of instances when some maximum time is exceeded. Such approaches—sometimes called *soft real-time*—arise when it is possible to miss the time constraint on an event occasionally, as long as not too many are missed. Real-time performance tends to be highly application-dependent.

Other key characteristics in many PMD applications are the need to minimize memory and the need to use energy efficiently. Energy efficiency is driven by both battery power and heat dissipation. The memory can be a substantial portion of the system cost, and it is important to optimize memory size in such cases. The importance of memory size translates to an emphasis on code size, since data size is dictated by the application.

Desktop Computing

The first, and possibly still the largest market in dollar terms, is desktop computing. Desktop computing spans from low-end netbooks that sell for under $300 to high-end, heavily configured workstations that may sell for $2500. Since 2008, more than half of the desktop computers made each year have been battery operated laptop computers. Desktop computing sales are declining.

Throughout this range in price and capability, the desktop market tends to be driven to optimize *price-performance.* This combination of performance (measured primarily in terms of compute performance and graphics performance) and price of a system is what matters most to customers in this market, and hence to computer designers. As a result, the newest, highest-performance microprocessors and cost-reduced microprocessors often appear first in desktop systems (see Section 1.6 for a discussion of the issues affecting the cost of computers).

Desktop computing also tends to be reasonably well characterized in terms of applications and benchmarking, though the increasing use of web-centric, interactive applications poses new challenges in performance evaluation.

Servers

As the shift to desktop computing occurred in the 1980s, the role of servers grew to provide larger-scale and more reliable file and computing services. Such servers have become the backbone of large-scale enterprise computing, replacing the traditional mainframe.

For servers, different characteristics are important. First, availability is critical. (We discuss availability in Section 1.7.) Consider the servers running ATM machines for banks or airline reservation systems. Failure of such server systems is far more catastrophic than failure of a single desktop, since these servers must operate seven days a week, 24 hours a day. Figure 1.3 estimates revenue costs of downtime for server applications.

Application	Cost of downtime per hour	Annual losses with downtime of		
		1% (87.6 h/year)	0.5% (43.8 h/year)	0.1% (8.8 h/year)
Brokerage service	$4,000,000	$350,400,000	$175,200,000	$35,000,000
Energy	$1,750,000	$153,300,000	$76,700,000	$15,300,000
Telecom	$1,250,000	$109,500,000	$54,800,000	$11,000,000
Manufacturing	$1,000,000	$87,600,000	$43,800,000	$8,800,000
Retail	$650,000	$56,900,000	$28,500,000	$5,700,000
Health care	$400,000	$35,000,000	$17,500,000	$3,500,000
Media	$50,000	$4,400,000	$2,200,000	$400,000

Figure 1.3 Costs rounded to nearest $100,000 of an unavailable system are shown by analyzing the cost of downtime (in terms of immediately lost revenue), assuming three different levels of availability, and that downtime is distributed uniformly. These data are from Landstrom (2014) and were collected and analyzed by Contingency Planning Research.

A second key feature of server systems is scalability. Server systems often grow in response to an increasing demand for the services they support or an expansion in functional requirements. Thus the ability to scale up the computing capacity, the memory, the storage, and the I/O bandwidth of a server is crucial.

Finally, servers are designed for efficient throughput. That is, the overall performance of the server—in terms of transactions per minute or web pages served per second—is what is crucial. Responsiveness to an individual request remains important, but overall efficiency and cost-effectiveness, as determined by how many requests can be handled in a unit time, are the key metrics for most servers. We return to the issue of assessing performance for different types of computing environments in Section 1.8.

Clusters/Warehouse-Scale Computers

The growth of Software as a Service (SaaS) for applications like search, social networking, video viewing and sharing, multiplayer games, online shopping, and so on has led to the growth of a class of computers called *clusters*. Clusters are collections of desktop computers or servers connected by local area networks to act as a single larger computer. Each node runs its own operating system, and nodes communicate using a networking protocol. WSCs are the largest of the clusters, in that they are designed so that tens of thousands of servers can act as one. Chapter 6 describes this class of extremely large computers.

Price-performance and power are critical to WSCs since they are so large. As Chapter 6 explains, the majority of the cost of a warehouse is associated with power and cooling of the computers inside the warehouse. The annual amortized computers themselves and the networking gear cost for a WSC is $40 million, because they are usually replaced every few years. When you are buying that

much computing, you need to buy wisely, because a 10% improvement in price-performance means an annual savings of $4 million (10% of $40 million) per WSC; a company like Amazon might have 100 WSCs!

WSCs are related to servers in that availability is critical. For example, Amazon.com had $136 billion in sales in 2016. As there are about 8800 hours in a year, the average revenue per hour was about $15 million. During a peak hour for Christmas shopping, the potential loss would be many times higher. As Chapter 6 explains, the difference between WSCs and servers is that WSCs use redundant, inexpensive components as the building blocks, relying on a software layer to catch and isolate the many failures that will happen with computing at this scale to deliver the availability needed for such applications. Note that scalability for a WSC is handled by the local area network connecting the computers and not by integrated computer hardware, as in the case of servers.

Supercomputers are related to WSCs in that they are equally expensive, costing hundreds of millions of dollars, but supercomputers differ by emphasizing floating-point performance and by running large, communication-intensive batch programs that can run for weeks at a time. In contrast, WSCs emphasize interactive applications, large-scale storage, dependability, and high Internet bandwidth.

Classes of Parallelism and Parallel Architectures

Parallelism at multiple levels is now the driving force of computer design across all four classes of computers, with energy and cost being the primary constraints. There are basically two kinds of parallelism in applications:

1. *Data-level parallelism (DLP)* arises because there are many data items that can be operated on at the same time.
2. *Task-level parallelism (TLP)* arises because tasks of work are created that can operate independently and largely in parallel.

Computer hardware in turn can exploit these two kinds of application parallelism in four major ways:

1. *Instruction-level parallelism* exploits data-level parallelism at modest levels with compiler help using ideas like pipelining and at medium levels using ideas like speculative execution.
2. *Vector architectures, graphic processor units (GPUs), and multimedia instruction sets* exploit data-level parallelism by applying a single instruction to a collection of data in parallel.
3. *Thread-level parallelism* exploits either data-level parallelism or task-level parallelism in a tightly coupled hardware model that allows for interaction between parallel threads.
4. *Request-level parallelism* exploits parallelism among largely decoupled tasks specified by the programmer or the operating system.

When Flynn (1966) studied the parallel computing efforts in the 1960s, he found a simple classification whose abbreviations we still use today. They target data-level parallelism and task-level parallelism. He looked at the parallelism in the instruction and data streams called for by the instructions at the most constrained component of the multiprocessor and placed all computers in one of four categories:

1. *Single instruction stream, single data stream* (SISD)—This category is the uniprocessor. The programmer thinks of it as the standard sequential computer, but it can exploit ILP. Chapter 3 covers SISD architectures that use ILP techniques such as superscalar and speculative execution.
2. *Single instruction stream, multiple data streams* (SIMD)—The same instruction is executed by multiple processors using different data streams. SIMD computers exploit *data-level parallelism* by applying the same operations to multiple items of data in parallel. Each processor has its own data memory (hence, the MD of SIMD), but there is a single instruction memory and control processor, which fetches and dispatches instructions. Chapter 4 covers DLP and three different architectures that exploit it: vector architectures, multimedia extensions to standard instruction sets, and GPUs.
3. *Multiple instruction streams, single data stream* (MISD)—No commercial multiprocessor of this type has been built to date, but it rounds out this simple classification.
4. *Multiple instruction streams, multiple data streams* (MIMD)—Each processor fetches its own instructions and operates on its own data, and it targets task-level parallelism. In general, MIMD is more flexible than SIMD and thus more generally applicable, but it is inherently more expensive than SIMD. For example, MIMD computers can also exploit data-level parallelism, although the overhead is likely to be higher than would be seen in an SIMD computer. This overhead means that grain size must be sufficiently large to exploit the parallelism efficiently. Chapter 5 covers tightly coupled MIMD architectures, which exploit *thread-level parallelism* because multiple cooperating threads operate in parallel. Chapter 6 covers loosely coupled MIMD architectures—specifically, *clusters* and *warehouse-scale computers*—that exploit *request-level parallelism*, where many independent tasks can proceed in parallel naturally with little need for communication or synchronization.

This taxonomy is a coarse model, as many parallel processors are hybrids of the SISD, SIMD, and MIMD classes. Nonetheless, it is useful to put a framework on the design space for the computers we will see in this book.

1.3 Defining Computer Architecture

The task the computer designer faces is a complex one: determine what attributes are important for a new computer, then design a computer to maximize

performance and energy efficiency while staying within cost, power, and availability constraints. This task has many aspects, including instruction set design, functional organization, logic design, and implementation. The implementation may encompass integrated circuit design, packaging, power, and cooling. Optimizing the design requires familiarity with a very wide range of technologies, from compilers and operating systems to logic design and packaging.

A few decades ago, the term *computer architecture* generally referred to only instruction set design. Other aspects of computer design were called *implementation,* often insinuating that implementation is uninteresting or less challenging.

We believe this view is incorrect. The architect's or designer's job is much more than instruction set design, and the technical hurdles in the other aspects of the project are likely more challenging than those encountered in instruction set design. We'll quickly review instruction set architecture before describing the larger challenges for the computer architect.

Instruction Set Architecture: The Myopic View of Computer Architecture

We use the term *instruction set architecture* (ISA) to refer to the actual programmer-visible instruction set in this book. The ISA serves as the boundary between the software and hardware. This quick review of ISA will use examples from 80x86, ARMv8, and RISC-V to illustrate the seven dimensions of an ISA. The most popular RISC processors come from ARM (Advanced RISC Machine), which were in 14.8 billion chips shipped in 2015, or roughly 50 times as many chips that shipped with 80x86 processors. Appendices A and K give more details on the three ISAs.

RISC-V ("RISC Five") is a modern RISC instruction set developed at the University of California, Berkeley, which was made free and openly adoptable in response to requests from industry. In addition to a full software stack (compilers, operating systems, and simulators), there are several RISC-V implementations freely available for use in custom chips or in field-programmable gate arrays. Developed 30 years after the first RISC instruction sets, RISC-V inherits its ancestors' good ideas—a large set of registers, easy-to-pipeline instructions, and a lean set of operations—while avoiding their omissions or mistakes. It is a free and open, elegant example of the RISC architectures mentioned earlier, which is why more than 60 companies have joined the RISC-V foundation, including AMD, Google, HP Enterprise, IBM, Microsoft, Nvidia, Qualcomm, Samsung, and Western Digital. We use the integer core ISA of RISC-V as the example ISA in this book.

1. *Class of ISA*—Nearly all ISAs today are classified as general-purpose register architectures, where the operands are either registers or memory locations. The 80x86 has 16 general-purpose registers and 16 that can hold floating-point data, while RISC-V has 32 general-purpose and 32 floating-point registers (see Figure 1.4). The two popular versions of this class are *register-memory* ISAs,

Register	Name	Use	Saver
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	–
x4	tp	Thread pointer	–
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP temporaries	Caller
f8-f9	fs0-fs1	FP saved registers	Callee
f10-f11	fa0-fa1	FP function arguments/return values	Caller
f12-f17	fa2-fa7	FP function arguments	Caller
f18-f27	fs2-fs11	FP saved registers	Callee
f28-f31	ft8-ft11	FP temporaries	Caller

Figure 1.4 RISC-V registers, names, usage, and calling conventions. In addition to the 32 general-purpose registers (x0–x31), RISC-V has 32 floating-point registers (f0–f31) that can hold either a 32-bit single-precision number or a 64-bit double-precision number. The registers that are preserved across a procedure call are labeled "Callee" saved.

such as the 80x86, which can access memory as part of many instructions, and *load-store* ISAs, such as ARMv8 and RISC-V, which can access memory only with load or store instructions. All ISAs announced since 1985 are load-store.

2. *Memory addressing*—Virtually all desktop and server computers, including the 80x86, ARMv8, and RISC-V, use byte addressing to access memory operands. Some architectures, like ARMv8, require that objects must be *aligned*. An access to an object of size s bytes at byte address A is aligned if A mod $s = 0$. (See Figure A.5 on page A-8.) The 80x86 and RISC-V do not require alignment, but accesses are generally faster if operands are aligned.
3. *Addressing modes*—In addition to specifying registers and constant operands, addressing modes specify the address of a memory object. RISC-V addressing modes are Register, Immediate (for constants), and Displacement, where a constant offset is added to a register to form the memory address. The 80x86 supports those three modes, plus three variations of displacement: no register (absolute), two registers (based indexed with displacement), and two registers

where one register is multiplied by the size of the operand in bytes (based with scaled index and displacement). It has more like the last three modes, minus the displacement field, plus register indirect, indexed, and based with scaled index. ARMv8 has the three RISC-V addressing modes plus PC-relative addressing, the sum of two registers, and the sum of two registers where one register is multiplied by the size of the operand in bytes. It also has autoincrement and autodecrement addressing, where the calculated address replaces the contents of one of the registers used in forming the address.

4. *Types and sizes of operands*—Like most ISAs, 80x86, ARMv8, and RISC-V support operand sizes of 8-bit (ASCII character), 16-bit (Unicode character or half word), 32-bit (integer or word), 64-bit (double word or long integer), and IEEE 754 floating point in 32-bit (single precision) and 64-bit (double precision). The 80x86 also supports 80-bit floating point (extended double precision).
5. *Operations*—The general categories of operations are data transfer, arithmetic logical, control (discussed next), and floating point. RISC-V is a simple and easy-to-pipeline instruction set architecture, and it is representative of the RISC architectures being used in 2017. Figure 1.5 summarizes the integer RISC-V ISA, and Figure 1.6 lists the floating-point ISA. The 80x86 has a much richer and larger set of operations (see Appendix K).
6. *Control flow instructions*—Virtually all ISAs, including these three, support conditional branches, unconditional jumps, procedure calls, and returns. All three use PC-relative addressing, where the branch address is specified by an address field that is added to the PC. There are some small differences. RISC-V conditional branches (`BE`, `BNE`, etc.) test the contents of registers, and the 80x86 and ARMv8 branches test condition code bits set as side effects of arithmetic/logic operations. The ARMv8 and RISC-V procedure call places the return address in a register, whereas the 80x86 call (`CALLF`) places the return address on a stack in memory.
7. *Encoding an ISA*—There are two basic choices on encoding: *fixed length* and *variable length*. All ARMv8 and RISC-V instructions are 32 bits long, which simplifies instruction decoding. Figure 1.7 shows the RISC-V instruction formats. The 80x86 encoding is variable length, ranging from 1 to 18 bytes. Variable-length instructions can take less space than fixed-length instructions, so a program compiled for the 80x86 is usually smaller than the same program compiled for RISC-V. Note that choices mentioned previously will affect how the instructions are encoded into a binary representation. For example, the number of registers and the number of addressing modes both have a significant impact on the size of instructions, because the register field and addressing mode field can appear many times in a single instruction. (Note that ARMv8 and RISC-V later offered extensions, called Thumb-2 and RV64IC, that provide a mix of 16-bit and 32-bit length instructions, respectively, to reduce program size. Code size for these compact versions of RISC architectures are smaller than that of the 80x86. See Appendix K.)

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 12-bit displacement + contents of a GPR*
`lb, lbu, sb`	Load byte, load byte unsigned, store byte (to/from integer registers)
`lh, lhu, sh`	Load half word, load half word unsigned, store half word (to/from integer registers)
`lw, lwu, sw`	Load word, load word unsigned, store word (to/from integer registers)
`ld, sd`	Load double word, store double word (to/from integer registers)
`flw, fld, fsw, fsd`	Load SP float, load DP float, store SP float, store DP float
`fmv._.x, fmv.x._`	Copy from/to integer register to/from floating-point register; "__" = S for single-precision, D for double-precision
`csrrw, csrrwi, csrrs, csrrsi, csrrc, csrrci`	Read counters and write status registers, which include counters: clock cycles, time, instructions retired
Arithmetic/logical	*Operations on integer or logical data in GPRs*
`add, addi, addw, addiw`	Add, add immediate (all immediates are 12 bits), add 32-bits only & sign-extend to 64 bits, add immediate 32-bits only
`sub, subw`	Subtract, subtract 32-bits only
`mul, mulw, mulh, mulhsu, mulhu`	Multiply, multiply 32-bits only, multiply upper half, multiply upper half signed-unsigned, multiply upper half unsigned
`div, divu, rem, remu`	Divide, divide unsigned, remainder, remainder unsigned
`divw, divuw, remw, remuw`	Divide and remainder: as previously, but divide only lower 32-bits, producing 32-bit sign-extended result
`and, andi`	And, and immediate
`or, ori, xor, xori`	Or, or immediate, exclusive or, exclusive or immediate
`lui`	Load upper immediate; loads bits 31-12 of register with immediate, then sign-extends
`auipc`	Adds immediate in bits 31–12 with zeros in lower bits to PC; used with `JALR` to transfer control to any 32-bit address
`sll, slli, srl, srli, sra, srai`	Shifts: shift left logical, right logical, right arithmetic; both variable and immediate forms
`sllw, slliw, srlw, srliw, sraw, sraiw`	Shifts: as previously, but shift lower 32-bits, producing 32-bit sign-extended result
`slt, slti, sltu, sltiu`	Set less than, set less than immediate, signed and unsigned
Control	*Conditional branches and jumps; PC-relative or through register*
`beq, bne, blt, bge, bltu, bgeu`	Branch GPR equal/not equal; less than; greater than or equal, signed and unsigned
`jal, jalr`	Jump and link: save PC + 4, target is PC-relative (`JAL`) or a register (`JALR`); if specify `x0` as destination register, then acts as a simple jump
`ecall`	Make a request to the supporting execution environment, which is usually an OS
`ebreak`	Debuggers used to cause control to be transferred back to a debugging environment
`fence, fence.i`	Synchronize threads to guarantee ordering of memory accesses; synchronize instructions and data for stores to instruction memory

Figure 1.5 Subset of the instructions in RISC-V. RISC-V has a base set of instructions (R64I) and offers optional extensions: multiply-divide (RVM), single-precision floating point (RVF), double-precision floating point (RVD). This figure includes RVM and the next one shows RVF and RVD. Appendix A gives much more detail on RISC-V.

Instruction type/opcode	Instruction meaning
Floating point	*FP operations on DP and SP formats*
`fadd.d`, `fadd.s`	Add DP, SP numbers
`fsub.d`, `fsub.s`	Subtract DP, SP numbers
`fmul.d`, `fmul.s`	Multiply DP, SP floating point
`fmadd.d`, `fmadd.s`, `fnmadd.d`, `fnmadd.s`	Multiply-add DP, SP numbers; negative multiply-add DP, SP numbers
`fmsub.d`, `fmsub.s`, `fnmsub.d`, `fnmsub.s`	Multiply-sub DP, SP numbers; negative multiply-sub DP, SP numbers
`fdiv.d`, `fdiv.s`	Divide DP, SP floating point
`fsqrt.d`, `fsqrt.s`	Square root DP, SP floating point
`fmax.d`, `fmax.s`, `fmin.d`, `fmin.s`	Maximum and minimum DP, SP floating point
`fcvt._._`, `fcvt._._u`, `fcvt._u._`	Convert instructions: `FCVT.x.y` converts from type `x` to type `y`, where `x` and `y` are `L` (64-bit integer), `W` (32-bit integer), `D` (DP), or `S` (SP). Integers can be unsigned (`U`)
`feq._`, `flt._`,`fle._`	Floating-point compare between floating-point registers and record the Boolean result in integer register; "`_`" = `S` for single-precision, `D` for double-precision
`fclass.d`, `fclass.s`	Writes to integer register a 10-bit mask that indicates the class of the floating-point number ($-\infty$, $+\infty$, -0, $+0$, NaN, …)
`fsgnj._`, `fsgnjn._`, `fsgnjx._`	Sign-injection instructions that changes only the sign bit: copy sign bit from other source, the oppositive of sign bit of other source, XOR of the 2 sign bits

Figure 1.6 Floating point instructions for RISC-V. RISC-V has a base set of instructions (R64I) and offers optional extensions for single-precision floating point (RVF) and double-precision floating point (RVD). SP = single precision; DP = double precision.

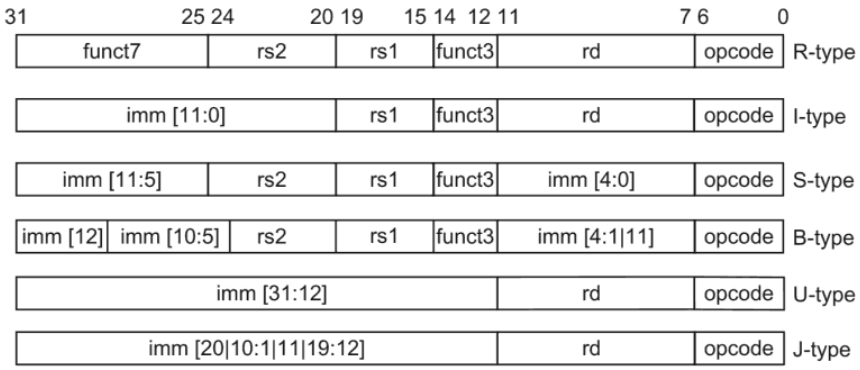


Figure 1.7 The base RISC-V instruction set architecture formats. All instructions are 32 bits long. The R format is for integer register-to-register operations, such as ADD, SUB, and so on. The I format is for loads and immediate operations, such as LD and ADDI. The B format is for branches and the J format is for jumps and link. The S format is for stores. Having a separate format for stores allows the three register specifiers (rd, rs1, rs2) to always be in the same location in all formats. The U format is for the wide immediate instructions (LUI, AUIPC).

The other challenges facing the computer architect beyond ISA design are particularly acute at the present, when the differences among instruction sets are small and when there are distinct application areas. Therefore, starting with the fourth edition of this book, beyond this quick review, the bulk of the instruction set material is found in the appendices (see Appendices A and K).

Genuine Computer Architecture: Designing the Organization and Hardware to Meet Goals and Functional Requirements

The implementation of a computer has two components: organization and hardware. The term *organization* includes the high-level aspects of a computer's design, such as the memory system, the memory interconnect, and the design of the internal processor or CPU (central processing unit—where arithmetic, logic, branching, and data transfer are implemented). The term *microarchitecture* is also used instead of organization. For example, two processors with the same instruction set architectures but different organizations are the AMD Opteron and the Intel Core i7. Both processors implement the 80x86 instruction set, but they have very different pipeline and cache organizations.

The switch to multiple processors per microprocessor led to the term *core* also being used for processors. Instead of saying multiprocessor microprocessor, the term *multicore* caught on. Given that virtually all chips have multiple processors, the term central processing unit, or CPU, is fading in popularity.

Hardware refers to the specifics of a computer, including the detailed logic design and the packaging technology of the computer. Often a line of computers contains computers with identical instruction set architectures and very similar organizations, but they differ in the detailed hardware implementation. For example, the Intel Core i7 (see Chapter 3) and the Intel Xeon E7 (see Chapter 5) are nearly identical but offer different clock rates and different memory systems, making the Xeon E7 more effective for server computers.

In this book, the word *architecture* covers all three aspects of computer design—instruction set architecture, organization or microarchitecture, and hardware.

Computer architects must design a computer to meet functional requirements as well as price, power, performance, and availability goals. Figure 1.8 summarizes requirements to consider in designing a new computer. Often, architects also must determine what the functional requirements are, which can be a major task. The requirements may be specific features inspired by the market. Application software typically drives the choice of certain functional requirements by determining how the computer will be used. If a large body of software exists for a particular instruction set architecture, the architect may decide that a new computer should implement an existing instruction set. The presence of a large market for a particular class of applications might encourage the designers to incorporate requirements that would make the computer competitive in that market. Later chapters examine many of these requirements and features in depth.

Functional requirements	Typical features required or supported
Application area	*Target of computer*
Personal mobile device	Real-time performance for a range of tasks, including interactive performance for graphics, video, and audio; energy efficiency (Chapters 2–5 and 7; Appendix A)
General-purpose desktop	Balanced performance for a range of tasks, including interactive performance for graphics, video, and audio (Chapters 2–5; Appendix A)
Servers	Support for databases and transaction processing; enhancements for reliability and availability; support for scalability (Chapters 2, 5, and 7; Appendices A, D, and F)
Clusters/warehouse-scale computers	Throughput performance for many independent tasks; error correction for memory; energy proportionality (Chapters 2, 6, and 7; Appendix F)
Internet of things/embedded computing	Often requires special support for graphics or video (or other application-specific extension); power limitations and power control may be required; real-time constraints (Chapters 2, 3, 5, and 7; Appendices A and E)
Level of software compatibility	*Determines amount of existing software for computer*
At programming language	Most flexible for designer; need new compiler (Chapters 3, 5, and 7; Appendix A)
Object code or binary compatible	Instruction set architecture is completely defined—little flexibility—but no investment needed in software or porting programs (Appendix A)
Operating system requirements	*Necessary features to support chosen OS* (Chapter 2; Appendix B)
Size of address space	Very important feature (Chapter 2); may limit applications
Memory management	Required for modern OS; may be paged or segmented (Chapter 2)
Protection	Different OS and application needs: page versus segment; virtual machines (Chapter 2)
Standards	*Certain standards may be required by marketplace*
Floating point	Format and arithmetic: IEEE 754 standard (Appendix J), special arithmetic for graphics or signal processing
I/O interfaces	For I/O devices: Serial ATA, Serial Attached SCSI, PCI Express (Appendices D and F)
Operating systems	UNIX, Windows, Linux, CISCO IOS
Networks	Support required for different networks: Ethernet, Infiniband (Appendix F)
Programming languages	Languages (ANSI C, C++, Java, Fortran) affect instruction set (Appendix A)

Figure 1.8 Summary of some of the most important functional requirements an architect faces. The left-hand column describes the class of requirement, while the right-hand column gives specific examples. The right-hand column also contains references to chapters and appendices that deal with the specific issues.

Architects must also be aware of important trends in both the technology and the use of computers because such trends affect not only the future cost but also the longevity of an architecture.

1.4 Trends in Technology

If an instruction set architecture is to prevail, it must be designed to survive rapid changes in computer technology. After all, a successful new instruction set

architecture may last decades—for example, the core of the IBM mainframe has been in use for more than 50 years. An architect must plan for technology changes that can increase the lifetime of a successful computer.

To plan for the evolution of a computer, the designer must be aware of rapid changes in implementation technology. Five implementation technologies, which change at a dramatic pace, are critical to modern implementations:

- *Integrated circuit logic technology*—Historically, transistor density increased by about 35% per year, quadrupling somewhat over four years. Increases in die size are less predictable and slower, ranging from 10% to 20% per year. The combined effect was a traditional growth rate in transistor count on a chip of about 40%–55% per year, or doubling every 18–24 months. This trend is popularly known as Moore's Law. Device speed scales more slowly, as we discuss below. Shockingly, Moore's Law is no more. The number of devices per chip is still increasing, but at a decelerating rate. Unlike in the Moore's Law era, we expect the doubling time to be stretched with each new technology generation.
- *Semiconductor DRAM* (dynamic random-access memory)—This technology is the foundation of main memory, and we discuss it in Chapter 2. The growth of DRAM has slowed dramatically, from quadrupling every three years as in the past. The 8-gigabit DRAM was shipping in 2014, but the 16-gigabit DRAM won't reach that state until 2019, and it looks like there will be no 32-gigabit DRAM (Kim, 2005). Chapter 2 mentions several other technologies that may replace DRAM when it hits its capacity wall.
- *Semiconductor Flash* (electrically erasable programmable read-only memory)—This nonvolatile semiconductor memory is the standard storage device in PMDs, and its rapidly increasing popularity has fueled its rapid growth rate in capacity. In recent years, the capacity per Flash chip increased by about 50%–60% per year, doubling roughly every 2 years. Currently, Flash memory is 8–10 times cheaper per bit than DRAM. Chapter 2 describes Flash memory.
- *Magnetic disk technology*—Prior to 1990, density increased by about 30% per year, doubling in three years. It rose to 60% per year thereafter, and increased to 100% per year in 1996. Between 2004 and 2011, it dropped back to about 40% per year, or doubled every two years. Recently, disk improvement has slowed to less than 5% per year. One way to increase disk capacity is to add more platters at the same areal density, but there are already seven platters within the one-inch depth of the 3.5-inch form factor disks. There is room for at most one or two more platters. The last hope for real density increase is to use a small laser on each disk read-write head to heat a 30 nm spot to 400°C so that it can be written magnetically before it cools. It is unclear whether Heat Assisted Magnetic Recording can be manufactured economically and reliably, although Seagate announced plans to ship HAMR in limited production in 2018. HAMR is the last chance for continued improvement in areal density of hard disk

drives, which are now 8–10 times cheaper per bit than Flash and 200–300 times cheaper per bit than DRAM. This technology is central to server- and warehouse-scale storage, and we discuss the trends in detail in Appendix D.

- *Network technology*—Network performance depends both on the performance of switches and on the performance of the transmission system. We discuss the trends in networking in Appendix F.

These rapidly changing technologies shape the design of a computer that, with speed and technology enhancements, may have a lifetime of 3–5 years. Key technologies such as Flash change sufficiently that the designer must plan for these changes. Indeed, designers often design for the next technology, knowing that, when a product begins shipping in volume, the following technology may be the most cost-effective or may have performance advantages. Traditionally, cost has decreased at about the rate at which density increases.

Although technology improves continuously, the impact of these increases can be in discrete leaps, as a threshold that allows a new capability is reached. For example, when MOS technology reached a point in the early 1980s where between 25,000 and 50,000 transistors could fit on a single chip, it became possible to build a single-chip, 32-bit microprocessor. By the late 1980s, first-level caches could go on a chip. By eliminating chip crossings within the processor and between the processor and the cache, a dramatic improvement in cost-performance and energy-performance was possible. This design was simply unfeasible until the technology reached a certain point. With multicore microprocessors and increasing numbers of cores each generation, even server computers are increasingly headed toward a single chip for all processors. Such technology thresholds are not rare and have a significant impact on a wide variety of design decisions.

Performance Trends: Bandwidth Over Latency

As we shall see in Section 1.8, *bandwidth* or *throughput* is the total amount of work done in a given time, such as megabytes per second for a disk transfer. In contrast, *latency* or *response time* is the time between the start and the completion of an event, such as milliseconds for a disk access. Figure 1.9 plots the relative improvement in bandwidth and latency for technology milestones for microprocessors, memory, networks, and disks. Figure 1.10 describes the examples and milestones in more detail.

Performance is the primary differentiator for microprocessors and networks, so they have seen the greatest gains: 32,000–40,000 × in bandwidth and 50–90 × in latency. Capacity is generally more important than performance for memory and disks, so capacity has improved more, yet bandwidth advances of 400–2400 × are still much greater than gains in latency of 8–9 ×.

Clearly, bandwidth has outpaced latency across these technologies and will likely continue to do so. A simple rule of thumb is that bandwidth grows by at least the square of the improvement in latency. Computer designers should plan accordingly.

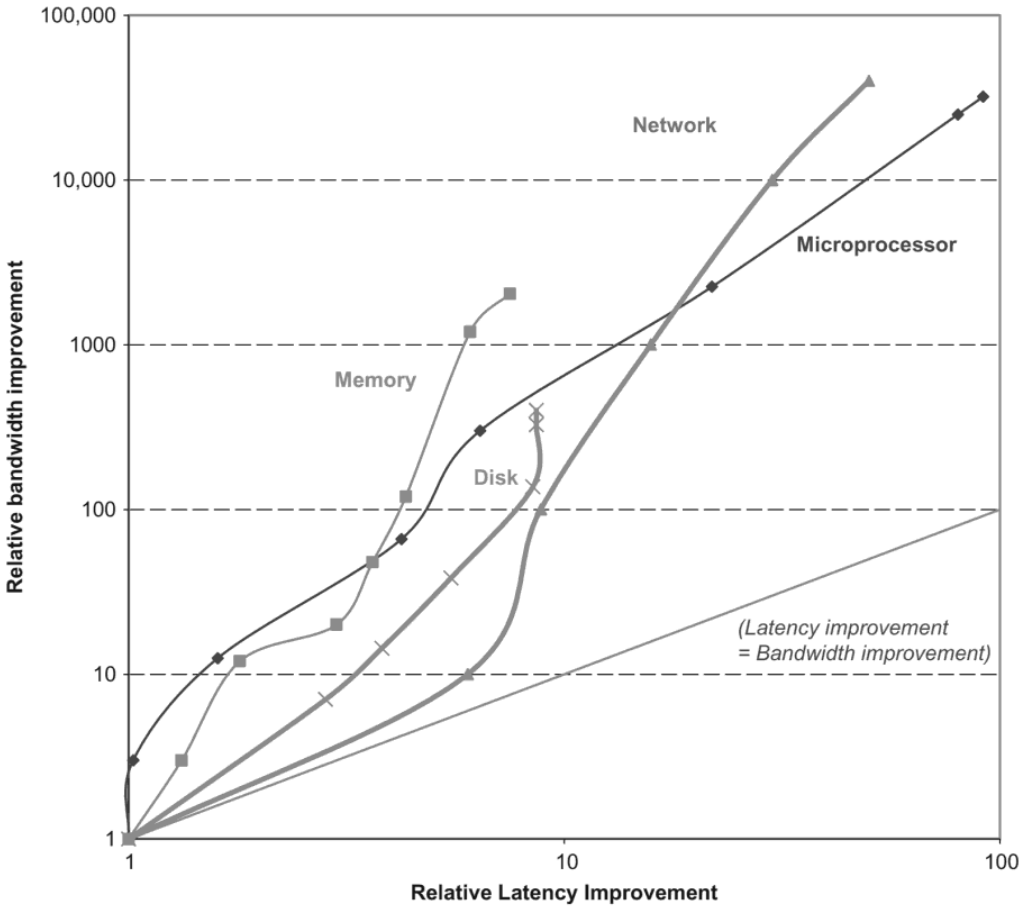


Figure 1.9 Log-log plot of bandwidth and latency milestones in Figure 1.10 relative to the first milestone. Note that latency improved 8–91 ×, while bandwidth improved about 400–32,000 ×. Except for networking, we note that there were modest improvements in latency and bandwidth in the other three technologies in the six years since the last edition: 0%–23% in latency and 23%–70% in bandwidth. Updated from Patterson, D., 2004. Latency lags bandwidth. Commun. ACM 47 (10), 71–75.

Scaling of Transistor Performance and Wires

Integrated circuit processes are characterized by the *feature size*, which is the minimum size of a transistor or a wire in either the x or y dimension. Feature sizes decreased from 10 μm in 1971 to 0.016 μm in 2017; in fact, we have switched units, so production in 2017 is referred to as "16 nm," and 7 nm chips are underway. Since the transistor count per square millimeter of silicon is determined by the surface area of a transistor, the density of transistors increases quadratically with a linear decrease in feature size.

Microprocessor	16-Bit address/ bus, microcoded	32-Bit address/ bus, microcoded	5-Stage pipeline, on-chip I & D caches, FPU	2-Way superscalar, 64-bit bus	Out-of-order 3-way superscalar	Out-of-order superpipelined, on-chip L2 cache	Multicore OOO 4-way on chip L3 cache, Turbo
Product	Intel 80286	Intel 80386	Intel 80486	Intel Pentium	Intel Pentium Pro	Intel Pentium 4	Intel Core i7
Year	1982	1985	1989	1993	1997	2001	2015
Die size (mm^2)	47	43	81	90	308	217	122
Transistors	134,000	275,000	1,200,000	3,100,000	5,500,000	42,000,000	1,750,000,000
Processors/chip	1	1	1	1	1	1	4
Pins	68	132	168	273	387	423	1400
Latency (clocks)	6	5	5	5	10	22	14
Bus width (bits)	16	32	32	64	64	64	196
Clock rate (MHz)	12.5	16	25	66	200	1500	4000
Bandwidth (MIPS)	2	6	25	132	600	4500	64,000
Latency (ns)	320	313	200	76	50	15	4
Memory module	DRAM	Page mode DRAM	Fast page mode DRAM	Fast page mode DRAM	Synchronous DRAM	Double data rate SDRAM	DDR4 SDRAM
Module width (bits)	16	16	32	64	64	64	64
Year	1980	1983	1986	1993	1997	2000	2016
Mbits/DRAM chip	0.06	0.25	1	16	64	256	4096
Die size (mm^2)	35	45	70	130	170	204	50
Pins/DRAM chip	16	16	18	20	54	66	134
Bandwidth (MBytes/s)	13	40	160	267	640	1600	27,000
Latency (ns)	225	170	125	75	62	52	30
Local area network	Ethernet	Fast Ethernet	Gigabit Ethernet	10 Gigabit Ethernet	100 Gigabit Ethernet	400 Gigabit Ethernet	
IEEE standard	802.3	803.3u	802.3ab	802.3ac	802.3ba	802.3bs	
Year	1978	1995	1999	2003	2010	2017	
Bandwidth (Mbits/seconds)	10	100	1000	10,000	100,000	400,000	
Latency (μs)	3000	500	340	190	100	60	
Hard disk	3600 RPM	5400 RPM	7200 RPM	10,000 RPM	15,000 RPM	15,000 RPM	
Product	CDC WrenI 94145-36	Seagate ST41600	Seagate ST15150	Seagate ST39102	Seagate ST373453	Seagate ST600MX0062	
Year	1983	1990	1994	1998	2003	2016	
Capacity (GB)	0.03	1.4	4.3	9.1	73.4	600	
Disk form factor	5.25 in.	5.25 in.	3.5 in.	3.5 in.	3.5 in.	3.5 in.	
Media diameter	5.25 in.	5.25 in.	3.5 in.	3.0 in.	2.5 in.	2.5 in.	
Interface	ST-412	SCSI	SCSI	SCSI	SCSI	SAS	
Bandwidth (MBytes/s)	0.6	4	9	24	86	250	
Latency (ms)	48.3	17.1	12.7	8.8	5.7	3.6	

Figure 1.10 Performance milestones over 25–40 years for microprocessors, memory, networks, and disks. The microprocessor milestones are several generations of IA-32 processors, going from a 16-bit bus, microcoded 80286 to a 64-bit bus, multicore, out-of-order execution, superpipelined Core i7. Memory module milestones go from 16-bit-wide, plain DRAM to 64-bit-wide double data rate version 3 synchronous DRAM. Ethernet advanced from 10 Mbits/s to 400 Gbits/s. Disk milestones are based on rotation speed, improving from 3600 to 15,000 RPM. Each case is best-case bandwidth, and latency is the time for a simple operation assuming no contention. Updated from Patterson, D., 2004. Latency lags bandwidth. Commun. ACM 47 (10), 71–75.

The increase in transistor performance, however, is more complex. As feature sizes shrink, devices shrink quadratically in the horizontal dimension and also shrink in the vertical dimension. The shrink in the vertical dimension requires a reduction in operating voltage to maintain correct operation and reliability of the transistors. This combination of scaling factors leads to a complex interrelationship between transistor performance and process feature size. To a first approximation, in the past the transistor performance improved linearly with decreasing feature size.

The fact that transistor count improves quadratically with a linear increase in transistor performance is both the challenge and the opportunity for which computer architects were created! In the early days of microprocessors, the higher rate of improvement in density was used to move quickly from 4-bit, to 8-bit, to 16-bit, to 32-bit, to 64-bit microprocessors. More recently, density improvements have supported the introduction of multiple processors per chip, wider SIMD units, and many of the innovations in speculative execution and caches found in Chapters 2–5.

Although transistors generally improve in performance with decreased feature size, wires in an integrated circuit do not. In particular, the signal delay for a wire increases in proportion to the product of its resistance and capacitance. Of course, as feature size shrinks, wires get shorter, but the resistance and capacitance per unit length get worse. This relationship is complex, since both resistance and capacitance depend on detailed aspects of the process, the geometry of a wire, the loading on a wire, and even the adjacency to other structures. There are occasional process enhancements, such as the introduction of copper, which provide one-time improvements in wire delay.

In general, however, wire delay scales poorly compared to transistor performance, creating additional challenges for the designer. In addition to the power dissipation limit, wire delay has become a major design obstacle for large integrated circuits and is often more critical than transistor switching delay. Larger and larger fractions of the clock cycle have been consumed by the propagation delay of signals on wires, but power now plays an even greater role than wire delay.

1.5 Trends in Power and Energy in Integrated Circuits

Today, energy is the biggest challenge facing the computer designer for nearly every class of computer. First, power must be brought in and distributed around the chip, and modern microprocessors use hundreds of pins and multiple interconnect layers just for power and ground. Second, power is dissipated as heat and must be removed.

Power and Energy: A Systems Perspective

How should a system architect or a user think about performance, power, and energy? From the viewpoint of a system designer, there are three primary concerns.

First, what is the maximum power a processor ever requires? Meeting this demand can be important to ensuring correct operation. For example, if a processor

attempts to draw more power than a power-supply system can provide (by drawing more current than the system can supply), the result is typically a voltage drop, which can cause devices to malfunction. Modern processors can vary widely in power consumption with high peak currents; hence they provide voltage indexing methods that allow the processor to slow down and regulate voltage within a wider margin. Obviously, doing so decreases performance.

Second, what is the sustained power consumption? This metric is widely called the *thermal design power* (TDP) because it determines the cooling requirement. TDP is neither peak power, which is often 1.5 times higher, nor is it the actual average power that will be consumed during a given computation, which is likely to be lower still. A typical power supply for a system is typically sized to exceed the TDP, and a cooling system is usually designed to match or exceed TDP. Failure to provide adequate cooling will allow the junction temperature in the processor to exceed its maximum value, resulting in device failure and possibly permanent damage. Modern processors provide two features to assist in managing heat, since the highest power (and hence heat and temperature rise) can exceed the long-term average specified by the TDP. First, as the thermal temperature approaches the junction temperature limit, circuitry lowers the clock rate, thereby reducing power. Should this technique not be successful, a second thermal overload trap is activated to power down the chip.

The third factor that designers and users need to consider is energy and energy efficiency. Recall that power is simply energy per unit time: 1 watt = 1 joule per second. Which metric is the right one for comparing processors: energy or power? In general, energy is always a better metric because it is tied to a specific task and the time required for that task. In particular, the energy to complete a workload is equal to the average power times the execution time for the workload.

Thus, if we want to know which of two processors is more efficient for a given task, we need to compare energy consumption (not power) for executing the task. For example, processor A may have a 20% higher average power consumption than processor B, but if A executes the task in only 70% of the time needed by B, its energy consumption will be $1.2 \times 0.7 = 0.84$, which is clearly better.

One might argue that in a large server or cloud, it is sufficient to consider the average power, since the workload is often assumed to be infinite, but this is misleading. If our cloud were populated with processor Bs rather than As, then the cloud would do less work for the same amount of energy expended. Using energy to compare the alternatives avoids this pitfall. Whenever we have a fixed workload, whether for a warehouse-size cloud or a smartphone, comparing energy will be the right way to compare computer alternatives, because the electricity bill for the cloud and the battery lifetime for the smartphone are both determined by the energy consumed.

When is power consumption a useful measure? The primary legitimate use is as a constraint: for example, an air-cooled chip might be limited to 100 W. It can be used as a metric if the workload is fixed, but then it's just a variation of the true metric of energy per task.

Energy and Power Within a Microprocessor

For CMOS chips, the traditional primary energy consumption has been in switching transistors, also called *dynamic energy*. The energy required per transistor is proportional to the product of the capacitive load driven by the transistor and the square of the voltage:

$$\text{Energy}_{\text{dynamic}} \propto \text{Capacitive load} \times \text{Voltage}^2$$

This equation is the energy of pulse of the logic transition of $0 \rightarrow 1 \rightarrow 0$ or $1 \rightarrow 0 \rightarrow 1$. The energy of a single transition ($0 \rightarrow 1$ or $1 \rightarrow 0$) is then:

$$\text{Energy}_{\text{dynamic}} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2$$

The power required per transistor is just the product of the energy of a transition multiplied by the frequency of transitions:

$$\text{Power}_{\text{dynamic}} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$$

For a fixed task, slowing clock rate reduces power, but not energy.

Clearly, dynamic power and energy are greatly reduced by lowering the voltage, so voltages have dropped from 5 V to just under 1 V in 20 years. The capacitive load is a function of the number of transistors connected to an output and the technology, which determines the capacitance of the wires and the transistors.

Example Some microprocessors today are designed to have adjustable voltage, so a 15% reduction in voltage may result in a 15% reduction in frequency. What would be the impact on dynamic energy and on dynamic power?

Answer Because the capacitance is unchanged, the answer for energy is the ratio of the voltages

$$\frac{\text{Energy}_{\text{new}}}{\text{Energy}_{\text{old}}} = \frac{(\text{Voltage} \times 0.85)^2}{\text{Voltage}^2} = 0.85^2 = 0.72$$

which reduces energy to about 72% of the original. For power, we add the ratio of the frequencies

$$\frac{\text{Power}_{\text{new}}}{\text{Power}_{\text{old}}} = 0.72 \times \frac{(\text{Frequency switched} \times 0.85)}{\text{Frequency switched}} = 0.61$$

shrinking power to about 61% of the original.

As we move from one process to the next, the increase in the number of transistors switching and the frequency with which they change dominate the decrease in load capacitance and voltage, leading to an overall growth in power consumption and energy. The first microprocessors consumed less than a watt, and the first

32-bit microprocessors (such as the Intel 80386) used about 2 W, whereas a 4.0 GHz Intel Core i7-6700K consumes 95 W. Given that this heat must be dissipated from a chip that is about 1.5 cm on a side, we are near the limit of what can be cooled by air, and this is where we have been stuck for nearly a decade.

Given the preceding equation, you would expect clock frequency growth to slow down if we can't reduce voltage or increase power per chip. Figure 1.11 shows that this has indeed been the case since 2003, even for the microprocessors in Figure 1.1 that were the highest performers each year. Note that this period of flatter clock rates corresponds to the period of slow performance improvement range in Figure 1.1.

Distributing the power, removing the heat, and preventing hot spots have become increasingly difficult challenges. Energy is now the major constraint to using transistors; in the past, it was the raw silicon area. Therefore modern

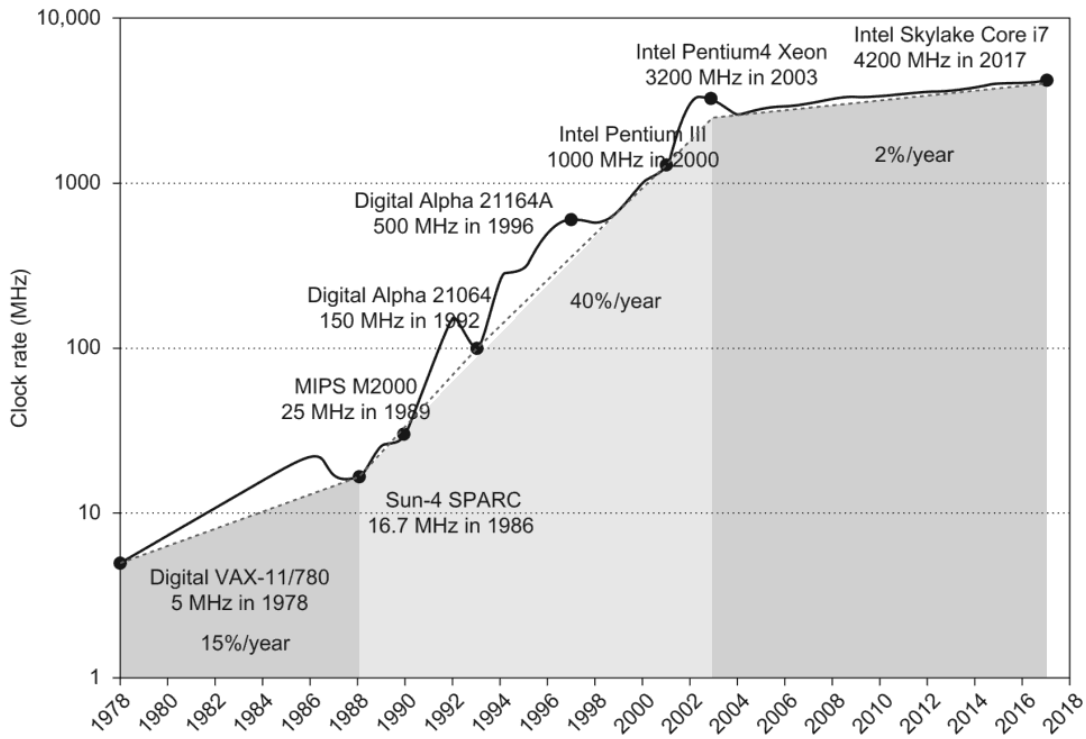


Figure 1.11 Growth in clock rate of microprocessors in Figure 1.1. Between 1978 and 1986, the clock rate improved less than 15% per year while performance improved by 22% per year. During the "renaissance period" of 52% performance improvement per year between 1986 and 2003, clock rates shot up almost 40% per year. Since then, the clock rate has been nearly flat, growing at less than 2% per year, while single processor performance improved recently at just 3.5% per year.

microprocessors offer many techniques to try to improve energy efficiency despite flat clock rates and constant supply voltages:

1. *Do nothing well.* Most microprocessors today turn off the clock of inactive modules to save energy and dynamic power. For example, if no floating-point instructions are executing, the clock of the floating-point unit is disabled. If some cores are idle, their clocks are stopped.
2. *Dynamic voltage-frequency scaling (DVFS).* The second technique comes directly from the preceding formulas. PMDs, laptops, and even servers have periods of low activity where there is no need to operate at the highest clock frequency and voltages. Modern microprocessors typically offer a few clock frequencies and voltages in which to operate that use lower power and energy. Figure 1.12 plots the potential power savings via DVFS for a server as the workload shrinks for three different clock rates: 2.4, 1.8, and 1 GHz. The overall server power savings is about 10%–15% for each of the two steps.
3. *Design for the typical case.* Given that PMDs and laptops are often idle, memory and storage offer low power modes to save energy. For example, DRAMs have a series of increasingly lower power modes to extend battery life in PMDs and laptops, and there have been proposals for disks that have a mode that spins more slowly when unused to save power. However, you cannot access DRAMs or disks in these modes, so you must return to fully active mode to read or write, no matter how low the access rate. As mentioned, microprocessors for PCs have been designed instead for heavy use at high operating temperatures, relying on on-chip temperature sensors to detect when activity should be reduced automatically to avoid overheating. This "emergency slowdown" allows manufacturers to design for a more typical case and then rely on this safety mechanism if someone really does run programs that consume much more power than is typical.

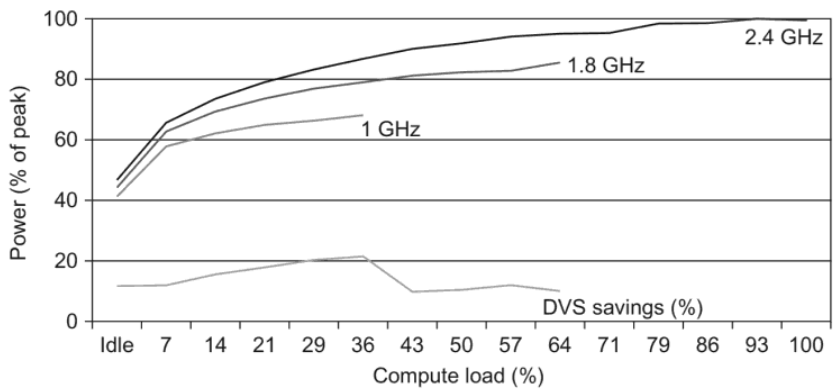


Figure 1.12 Energy savings for a server using an AMD Opteron microprocessor, 8 GB of DRAM, and one ATA disk. At 1.8 GHz, the server can handle at most up to two-thirds of the workload without causing service-level violations, and at 1 GHz, it can safely handle only one-third of the workload (Figure 5.11 in Barroso and Hölzle, 2009).

4. *Overclocking*. Intel started offering *Turbo mode* in 2008, where the chip decides that it is safe to run at a higher clock rate for a short time, possibly on just a few cores, until temperature starts to rise. For example, the 3.3 GHz Core i7 can run in short bursts for 3.6 GHz. Indeed, the highest-performing microprocessors each year since 2008 shown in Figure 1.1 have all offered temporary overclocking of about 10% over the nominal clock rate. For single-threaded code, these microprocessors can turn off all cores but one and run it faster. Note that, although the operating system can turn off Turbo mode, there is no notification once it is enabled, so the programmers may be surprised to see their programs vary in performance because of room temperature!

Although dynamic power is traditionally thought of as the primary source of power dissipation in CMOS, static power is becoming an important issue because leakage current flows even when a transistor is off:

$$\text{Power}_{\text{static}} \propto \text{Current}_{\text{static}} \times \text{Voltage}$$

That is, static power is proportional to the number of devices.

Thus increasing the number of transistors increases power even if they are idle, and current leakage increases in processors with smaller transistor sizes. As a result, very low-power systems are even turning off the power supply (*power gating*) to inactive modules in order to control loss because of leakage. In 2011 the goal for leakage was 25% of the total power consumption, with leakage in high-performance designs sometimes far exceeding that goal. Leakage can be as high as 50% for such chips, in part because of the large SRAM caches that need power to maintain the storage values. (The S in SRAM is for static.) The only hope to stop leakage is to turn off power to the chips' subsets.

Finally, because the processor is just a portion of the whole energy cost of a system, it can make sense to use a faster, less energy-efficient processor to allow the rest of the system to go into a sleep mode. This strategy is known as *race-to-halt*.

The importance of power and energy has increased the scrutiny on the efficiency of an innovation, so the primary evaluation now is tasks per joule or performance per watt, contrary to performance per mm^2 of silicon as in the past. This new metric affects approaches to parallelism, as we will see in Chapters 4 and 5.

The Shift in Computer Architecture Because of Limits of Energy

As transistor improvement decelerates, computer architects must look elsewhere for improved energy efficiency. Indeed, given the energy budget, it is easy today to design a microprocessor with so many transistors that they cannot all be turned on at the same time. This phenomenon has been called *dark silicon*, in that much of a chip cannot be unused ("dark") at any moment in time because of thermal constraints. This observation has led architects to reexamine the fundamentals of processors' design in the search for a greater energy-cost performance.

Figure 1.13, which lists the energy cost and area cost of the building blocks of a modern computer, reveals surprisingly large ratios. For example, a 32-bit

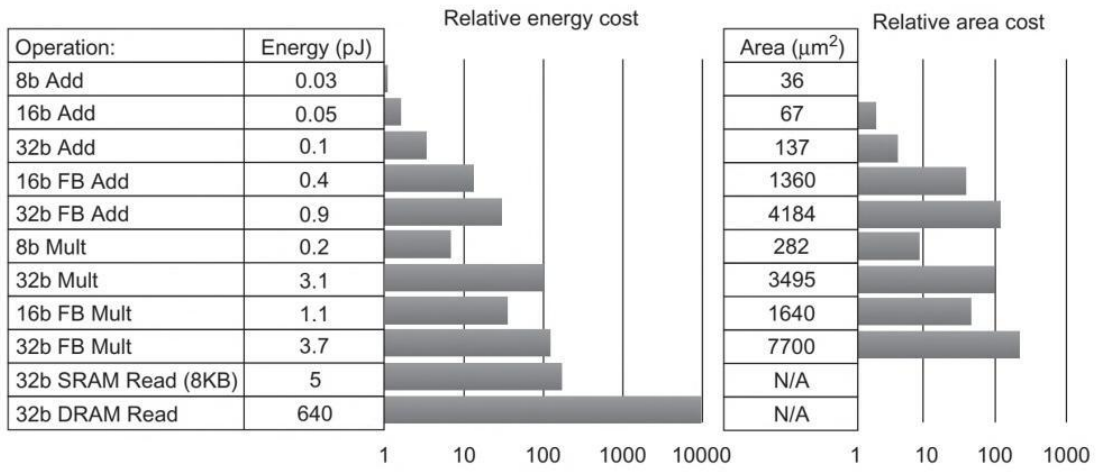


Figure 1.13 Comparison of the energy and die area of arithmetic operations and energy cost of accesses to SRAM and DRAM. [Azizi][Dally]. Area is for TSMC 45 nm technology node.

floating-point addition uses 30 times as much energy as an 8-bit integer add. The area difference is even larger, by 60 times. However, the biggest difference is in memory; a 32-bit DRAM access takes 20,000 times as much energy as an 8-bit addition. A small SRAM is 125 times more energy-efficient than DRAM, which demonstrates the importance of careful uses of caches and memory buffers.

The new design principle of minimizing energy per task combined with the relative energy and area costs in Figure 1.13 have inspired a new direction for computer architecture, which we describe in Chapter 7. Domain-specific processors save energy by reducing wide floating-point operations and deploying special-purpose memories to reduce accesses to DRAM. They use those saving to provide 10–100 more (narrower) integer arithmetic units than a traditional processor. Although such processors perform only a limited set of tasks, they perform them remarkably faster and more energy efficiently than a general-purpose processor.

Like a hospital with general practitioners and medical specialists, computers in this energy-aware world will likely be combinations of general-purpose cores that can perform any task and special-purpose cores that do a few things extremely well and even more cheaply.

1.6 Trends in Cost

Although costs tend to be less important in some computer designs—specifically supercomputers—cost-sensitive designs are of growing significance. Indeed, in the past 35 years, the use of technology improvements to lower cost, as well as increase performance, has been a major theme in the computer industry.

Textbooks often ignore the cost half of cost-performance because costs change, thereby dating books, and because the issues are subtle and differ across industry segments. Nevertheless, it's essential for computer architects to have an understanding of cost and its factors in order to make intelligent decisions about whether a new feature should be included in designs where cost is an issue. (Imagine architects designing skyscrapers without any information on costs of steel beams and concrete!)

This section discusses the major factors that influence the cost of a computer and how these factors are changing over time.

The Impact of Time, Volume, and Commoditization

The cost of a manufactured computer component decreases over time even without significant improvements in the basic implementation technology. The underlying principle that drives costs down is the *learning curve*—manufacturing costs decrease over time. The learning curve itself is best measured by change in *yield*—the percentage of manufactured devices that survives the testing procedure. Whether it is a chip, a board, or a system, designs that have twice the yield will have half the cost.

Understanding how the learning curve improves yield is critical to projecting costs over a product's life. One example is that the price per megabyte of DRAM has dropped over the long term. Since DRAMs tend to be priced in close relationship to cost—except for periods when there is a shortage or an oversupply—price and cost of DRAM track closely.

Microprocessor prices also drop over time, but because they are less standardized than DRAMs, the relationship between price and cost is more complex. In a period of significant competition, price tends to track cost closely, although microprocessor vendors probably rarely sell at a loss.

Volume is a second key factor in determining cost. Increasing volumes affect cost in several ways. First, they decrease the time needed to get through the learning curve, which is partly proportional to the number of systems (or chips) manufactured. Second, volume decreases cost because it increases purchasing and manufacturing efficiency. As a rule of thumb, some designers have estimated that costs decrease about 10% for each doubling of volume. Moreover, volume decreases the amount of development costs that must be amortized by each computer, thus allowing cost and selling price to be closer and still make a profit.

Commodities are products that are sold by multiple vendors in large volumes and are essentially identical. Virtually all the products sold on the shelves of grocery stores are commodities, as are standard DRAMs, Flash memory, monitors, and keyboards. In the past 30 years, much of the personal computer industry has become a commodity business focused on building desktop and laptop computers running Microsoft Windows.

Because many vendors ship virtually identical products, the market is highly competitive. Of course, this competition decreases the gap between cost and selling

price, but it also decreases cost. Reductions occur because a commodity market has both volume and a clear product definition, which allows multiple suppliers to compete in building components for the commodity product. As a result, the overall product cost is lower because of the competition among the suppliers of the components and the volume efficiencies the suppliers can achieve. This rivalry has led to the low end of the computer business being able to achieve better price-performance than other sectors and has yielded greater growth at the low end, although with very limited profits (as is typical in any commodity business).

Cost of an Integrated Circuit

Why would a computer architecture book have a section on integrated circuit costs? In an increasingly competitive computer marketplace where standard parts—disks, Flash memory, DRAMs, and so on—are becoming a significant portion of any system's cost, integrated circuit costs are becoming a greater portion of the cost that varies between computers, especially in the high-volume, cost-sensitive portion of the market. Indeed, with PMDs' increasing reliance of whole *systems on a chip* (SOC), the cost of the integrated circuits is much of the cost of the PMD. Thus computer designers must understand the costs of chips in order to understand the costs of current computers.

Although the costs of integrated circuits have dropped exponentially, the basic process of silicon manufacture is unchanged: A *wafer* is still tested and chopped into *dies* that are packaged (see Figures 1.14–1.16). Therefore the cost of a packaged integrated circuit is

$$\text{Cost of integrated circuit} = \frac{\text{Cost of die} + \text{Cost of testing die} + \text{Cost of packaging and final test}}{\text{Final test yield}}$$

In this section, we focus on the cost of dies, summarizing the key issues in testing and packaging at the end.

Learning how to predict the number of good chips per wafer requires first learning how many dies fit on a wafer and then learning how to predict the percentage of those that will work. From there it is simple to predict cost:

$$\text{Cost of die} = \frac{\text{Cost of wafer}}{\text{Dies per wafer} \times \text{Die yield}}$$

The most interesting feature of this initial term of the chip cost equation is its sensitivity to die size, shown below.

The number of dies per wafer is approximately the area of the wafer divided by the area of the die. It can be more accurately estimated by

$$\text{Dies per wafer} = \frac{\pi \times (\text{Wafer diameter}/2)^2}{\text{Die area}} - \frac{\pi \times \text{Wafer diameter}}{\sqrt{2 \times \text{Die area}}}$$

The first term is the ratio of wafer area (πr^2) to die area. The second compensates for the "square peg in a round hole" problem—rectangular dies near the periphery

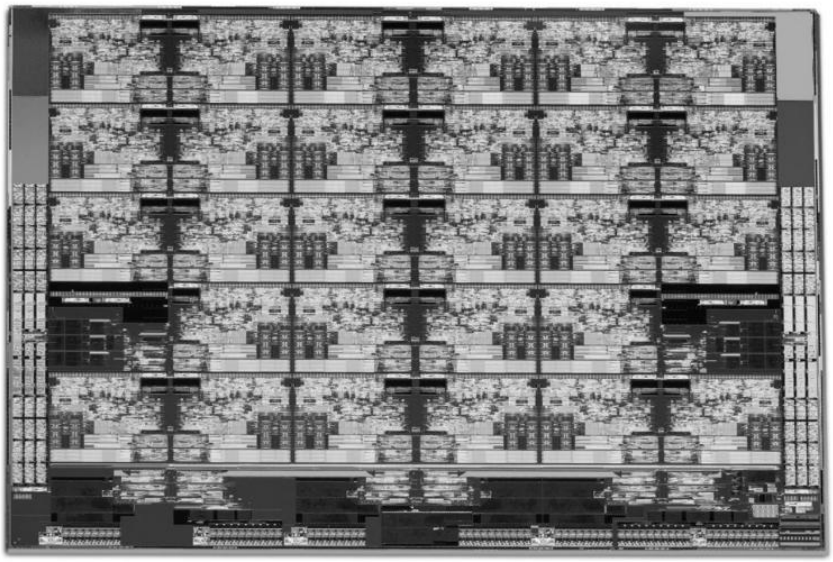


Figure 1.14 Photograph of an Intel Skylake microprocessor die, which is evaluated in Chapter 4.

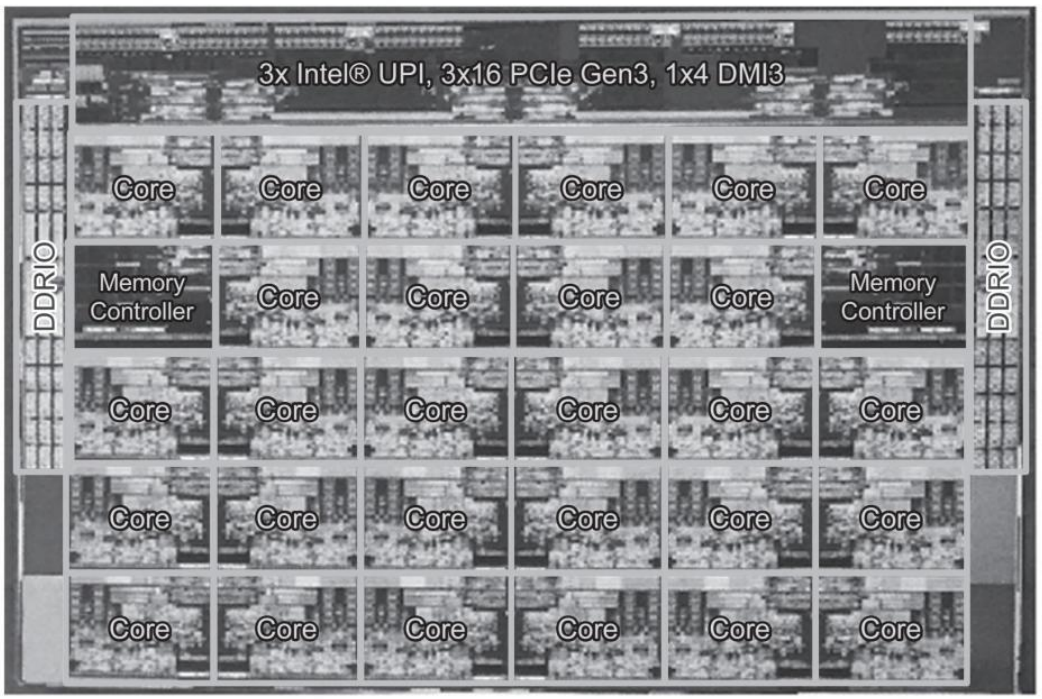


Figure 1.15 The components of the microprocessor die in Figure 1.14 are labeled with their functions.

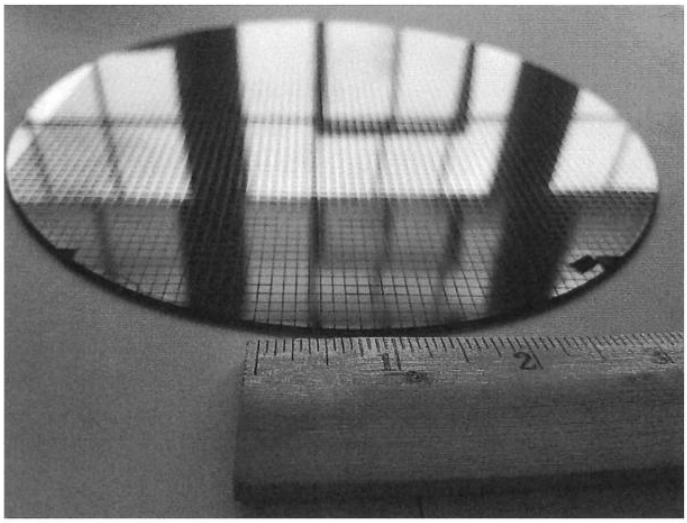


Figure 1.16 This 200 mm diameter wafer of RISC-V dies was designed by SiFive. It has two types of RISC-V dies using an older, larger processing line. An FE310 die is 2.65 mm × 2.72 mm and an SiFive test die that is 2.89 mm × 2.72 mm. The wafer contains 1846 of the former and 1866 of the latter, totaling 3712 chips.

of round wafers. Dividing the circumference (πd) by the diagonal of a square die is approximately the number of dies along the edge.

Example Find the number of dies per 300 mm (30 cm) wafer for a die that is 1.5 cm on a side and for a die that is 1.0 cm on a side.

Answer When die area is 2.25 cm^2:

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{2.25} - \frac{\pi \times 30}{\sqrt{2 \times 2.25}} = \frac{706.9}{2.25} - \frac{94.2}{2.12} = 270$$

Because the area of the larger die is 2.25 times bigger, there are roughly 2.25 as many smaller dies per wafer:

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{1.00} - \frac{\pi \times 30}{\sqrt{2 \times 1.00}} = \frac{706.9}{1.00} - \frac{94.2}{1.41} = 640$$

However, this formula gives only the maximum number of dies per wafer. The critical question is: What is the fraction of *good* dies on a wafer, or the *die yield*? A simple model of integrated circuit yield, which assumes that defects are randomly

distributed over the wafer and that yield is inversely proportional to the complexity of the fabrication process, leads to the following:

$$\text{Die yield} = \text{Wafer yield} \times 1/(1 + \text{Defects per unit area} \times \text{Die area})^N$$

This Bose-Einstein formula is an empirical model developed by looking at the yield of many manufacturing lines (Sydow, 2006), and it still applies today. *Wafer yield* accounts for wafers that are completely bad and so need not be tested. For simplicity, we'll just assume the wafer yield is 100%. Defects per unit area is a measure of the random manufacturing defects that occur. In 2017 the value was typically 0.08–0.10 defects per square inch for a 28-nm node and 0.10–0.30 for the newer 16 nm node because it depends on the maturity of the process (recall the learning curve mentioned earlier). The metric versions are 0.012–0.016 defects per square centimeter for 28 nm and 0.016–0.047 for 16 nm. Finally, N is a parameter called the process-complexity factor, a measure of manufacturing difficulty. For 28 nm processes in 2017, N is 7.5–9.5. For a 16 nm process, N ranges from 10 to 14.

Example Find the die yield for dies that are 1.5 cm on a side and 1.0 cm on a side, assuming a defect density of 0.047 per cm^2 and N is 12.

Answer The total die areas are 2.25 and 1.00 cm^2. For the larger die, the yield is

$$\text{Die yield} = 1/(1 + 0.047 \times 2.25)^{12} \times 270 = 120$$

For the smaller die, the yield is

$$\text{Die yield} = 1/(1 + 0.047 \times 1.00)^{12} \times 640 = 444$$

The bottom line is the number of good dies per wafer. Less than half of all the large dies are good, but nearly 70% of the small dies are good.

Although many microprocessors fall between 1.00 and 2.25 cm^2, low-end embedded 32-bit processors are sometimes as small as 0.05 cm^2, processors used for embedded control (for inexpensive IoT devices) are often less than 0.01 cm^2, and high-end server and GPU chips can be as large as 8 cm^2.

Given the tremendous price pressures on commodity products such as DRAM and SRAM, designers have included redundancy as a way to raise yield. For a number of years, DRAMs have regularly included some redundant memory cells so that a certain number of flaws can be accommodated. Designers have used similar techniques in both standard SRAMs and in large SRAM arrays used for caches within microprocessors. GPUs have 4 redundant processors out of 84 for the same reason. Obviously, the presence of redundant entries can be used to boost the yield significantly.

In 2017 processing of a 300 mm (12-inch) diameter wafer in a 28-nm technology costs between $4000 and $5000, and a 16-nm wafer costs about $7000. Assuming a processed wafer cost of $7000, the cost of the 1.00 cm^2 die would be around $16, but the cost per die of the 2.25 cm^2 die would be about $58, or almost four times the cost of a die that is a little over twice as large.

What should a computer designer remember about chip costs? The manufacturing process dictates the wafer cost, wafer yield, and defects per unit area, so the sole control of the designer is die area. In practice, because the number of defects per unit area is small, the number of good dies per wafer, and therefore the cost per die, grows roughly as the square of the die area. The computer designer affects die size, and thus cost, both by what functions are included on or excluded from the die and by the number of I/O pins.

Before we have a part that is ready for use in a computer, the die must be tested (to separate the good dies from the bad), packaged, and tested again after packaging. These steps all add significant costs, increasing the total by half.

The preceding analysis focused on the variable costs of producing a functional die, which is appropriate for high-volume integrated circuits. There is, however, one very important part of the fixed costs that can significantly affect the cost of an integrated circuit for low volumes (less than 1 million parts), namely, the cost of a mask set. Each step in the integrated circuit process requires a separate mask. Therefore, for modern high-density fabrication processes with up to 10 metal layers, mask costs are about $4 million for 16 nm and $1.5 million for 28 nm.

The good news is that semiconductor companies offer "shuttle runs" to dramatically lower the costs of tiny test chips. They lower costs by putting many small designs onto a single die to amortize the mask costs, and then later split the dies into smaller pieces for each project. Thus TSMC delivers 80–100 untested dies that are 1.57 × 1.57 mm in a 28 nm process for $30,000 in 2017. Although these die are tiny, they offer the architect millions of transistors to play with. For example, several RISC-V processors would fit on such a die.

Although shuttle runs help with prototyping and debugging runs, they don't address small-volume production of tens to hundreds of thousands of parts. Because mask costs are likely to continue to increase, some designers are incorporating reconfigurable logic to enhance the flexibility of a part and thus reduce the cost implications of masks.

Cost Versus Price

With the commoditization of computers, the margin between the cost to manufacture a product and the price the product sells for has been shrinking. Those margins pay for a company's research and development (R&D), marketing, sales, manufacturing equipment maintenance, building rental, cost of financing, pretax profits, and taxes. Many engineers are surprised to find that most companies spend only 4% (in the commodity PC business) to 12% (in the high-end server business) of their income on R&D, which includes all engineering.

Cost of Manufacturing Versus Cost of Operation

For the first four editions of this book, cost meant the cost to build a computer and price meant price to purchase a computer. With the advent of WSCs, which contain tens of thousands of servers, the cost to operate the computers is significant in addition to the cost of purchase. Economists refer to these two costs as capital expenses (CAPEX) and operational expenses (OPEX).

As Chapter 6 shows, the amortized purchase price of servers and networks is about half of the monthly cost to operate a WSC, assuming a short lifetime of the IT equipment of 3–4 years. About 40% of the monthly operational costs are for power use and the amortized infrastructure to distribute power and to cool the IT equipment, despite this infrastructure being amortized over 10–15 years. Thus, to lower operational costs in a WSC, computer architects need to use energy efficiently.

1.7 Dependability

Historically, integrated circuits were one of the most reliable components of a computer. Although their pins may be vulnerable, and faults may occur over communication channels, the failure rate inside the chip was very low. That conventional wisdom is changing as we head to feature sizes of 16 nm and smaller, because both transient faults and permanent faults are becoming more commonplace, so architects must design systems to cope with these challenges. This section gives a quick overview of the issues in dependability, leaving the official definition of the terms and approaches to Section D.3 in Appendix D.

Computers are designed and constructed at different layers of abstraction. We can descend recursively down through a computer seeing components enlarge themselves to full subsystems until we run into individual transistors. Although some faults are widespread, like the loss of power, many can be limited to a single component in a module. Thus utter failure of a module at one level may be considered merely a component error in a higher-level module. This distinction is helpful in trying to find ways to build dependable computers.

One difficult question is deciding when a system is operating properly. This theoretical point became concrete with the popularity of Internet services. Infrastructure providers started offering *service level agreements* (SLAs) or *service level objectives* (SLOs) to guarantee that their networking or power service would be dependable. For example, they would pay the customer a penalty if they did not meet an agreement of some hours per month. Thus an SLA could be used to decide whether the system was up or down.

Systems alternate between two states of service with respect to an SLA:

1. *Service accomplishment*, where the service is delivered as specified.
2. *Service interruption*, where the delivered service is different from the SLA.

Transitions between these two states are caused by *failures* (from state 1 to state 2) or *restorations* (2 to 1). Quantifying these transitions leads to the two main measures of dependability:

- *Module reliability* is a measure of the continuous service accomplishment (or, equivalently, of the time to failure) from a reference initial instant. Therefore the *mean time to failure* (MTTF) is a reliability measure. The reciprocal of MTTF is a rate of failures, generally reported as failures per billion hours of operation, or *FIT* (for *failures in time*). Thus an MTTF of 1,000,000 hours equals $10^9/10^6$ or 1000 FIT. Service interruption is measured as *mean time to repair* (MTTR). *Mean time between failures* (MTBF) is simply the sum of MTTF + MTTR. Although MTBF is widely used, MTTF is often the more appropriate term. If a collection of modules has exponentially distributed lifetimes—meaning that the age of a module is not important in probability of failure—the overall failure rate of the collection is the sum of the failure rates of the modules.
- *Module availability* is a measure of the service accomplishment with respect to the alternation between the two states of accomplishment and interruption. For nonredundant systems with repair, module availability is

$$\text{Module availability} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$

Note that reliability and availability are now quantifiable metrics, rather than synonyms for dependability. From these definitions, we can estimate reliability of a system quantitatively if we make some assumptions about the reliability of components and that failures are independent.

Example Assume a disk subsystem with the following components and MTTF:

- 10 disks, each rated at 1,000,000-hour MTTF
- 1 ATA controller, 500,000-hour MTTF
- 1 power supply, 200,000-hour MTTF
- 1 fan, 200,000-hour MTTF
- 1 ATA cable, 1,000,000-hour MTTF

Using the simplifying assumptions that the lifetimes are exponentially distributed and that failures are independent, compute the MTTF of the system as a whole.

Answer The sum of the failure rates is

$$\text{Failure rate}_{\text{system}} = 10 \times \frac{1}{1{,}000{,}000} + \frac{1}{500{,}000} + \frac{1}{200{,}000} + \frac{1}{200{,}000} + \frac{1}{1{,}000{,}000}$$

$$= \frac{10 + 2 + 5 + 5 + 1}{1{,}000{,}000 \text{ hours}} = \frac{23}{1{,}000{,}000} = \frac{23{,}000}{1{,}000{,}000{,}000 \text{ hours}}$$

or 23,000 FIT. The MTTF for the system is just the inverse of the failure rate

$$\text{MTTF}_{\text{system}} = \frac{1}{\text{Failure rate}_{\text{system}}} = \frac{1{,}000{,}000{,}000 \text{ hours}}{23{,}000} = 43{,}500 \text{ hours}$$

or just under 5 years.

The primary way to cope with failure is redundancy, either in time (repeat the operation to see if it still is erroneous) or in resources (have other components to take over from the one that failed). Once the component is replaced and the system is fully repaired, the dependability of the system is assumed to be as good as new. Let's quantify the benefits of redundancy with an example.

Example Disk subsystems often have redundant power supplies to improve dependability. Using the preceding components and MTTFs, calculate the reliability of redundant power supplies. Assume that one power supply is sufficient to run the disk subsystem and that we are adding one redundant power supply.

Answer We need a formula to show what to expect when we can tolerate a failure and still provide service. To simplify the calculations, we assume that the lifetimes of the components are exponentially distributed and that there is no dependency between the component failures. MTTF for our redundant power supplies is the mean time until one power supply fails divided by the chance that the other will fail before the first one is replaced. Thus, if the chance of a second failure before repair is small, then the MTTF of the pair is large.

Since we have two power supplies and independent failures, the mean time until one supply fails is $\text{MTTF}_{\text{power supply}}/2$. A good approximation of the probability of a second failure is MTTR over the mean time until the other power supply fails. Therefore a reasonable approximation for a redundant pair of power supplies is

$$\text{MTTF}_{\text{power supply pair}} = \frac{\text{MTTF}_{\text{power supply}}/2}{\frac{\text{MTTR}_{\text{power supply}}}{\text{MTTF}_{\text{power supply}}}} = \frac{\text{MTTF}^2_{\text{power supply}}/2}{\text{MTTR}_{\text{power supply}}} = \frac{\text{MTTF}^2_{\text{power supply}}}{2 \times \text{MTTR}_{\text{power supply}}}$$

Using the preceding MTTF numbers, if we assume it takes on average 24 hours for a human operator to notice that a power supply has failed and to replace it, the reliability of the fault tolerant pair of power supplies is

$$\text{MTTF}_{\text{power supply pair}} = \frac{\text{MTTF}^2_{\text{power supply}}}{2 \times \text{MTTR}_{\text{power supply}}} = \frac{200{,}000^2}{2 \times 24} \cong 830{,}000{,}000$$

making the pair about 4150 times more reliable than a single power supply.

Having quantified the cost, power, and dependability of computer technology, we are ready to quantify performance.

1.8 Measuring, Reporting, and Summarizing Performance

When we say one computer is faster than another one is, what do we mean? The user of a cell phone may say a computer is faster when a program runs in less time, while an Amazon.com administrator may say a computer is faster when it completes more transactions per hour. The cell phone user wants to reduce *response time*—the time between the start and the completion of an event—also referred to as *execution time*. The operator of a WSC wants to increase *throughput*—the total amount of work done in a given time.

In comparing design alternatives, we often want to relate the performance of two different computers, say, X and Y. The phrase "X is faster than Y" is used here to mean that the response time or execution time is lower on X than on Y for the given task. In particular, "X is n times as fast as Y" will mean

$$\frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Since execution time is the reciprocal of performance, the following relationship holds:

$$n = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = \frac{\dfrac{1}{\text{Performance}_Y}}{\dfrac{1}{\text{Performance}_X}} = \frac{\text{Performance}_X}{\text{Performance}_Y}$$

The phrase "the throughput of X is 1.3 times as fast as Y" signifies here that the number of tasks completed per unit time on computer X is 1.3 times the number completed on Y.

Unfortunately, time is not always the metric quoted in comparing the performance of computers. Our position is that the only consistent and reliable measure of performance is the execution time of real programs, and that all proposed alternatives to time as the metric or to real programs as the items measured have eventually led to misleading claims or even mistakes in computer design.

Even execution time can be defined in different ways depending on what we count. The most straightforward definition of time is called *wall-clock time*, *response time*, or *elapsed time*, which is the latency to complete a task, including storage accesses, memory accesses, input/output activities, operating system overhead—everything. With multiprogramming, the processor works on another program while waiting for I/O and may not necessarily minimize the elapsed time of one program. Thus we need a term to consider this activity. *CPU time* recognizes this distinction and means the time the processor is computing, *not* including the time waiting for I/O or running other programs. (Clearly, the response time seen by the user is the elapsed time of the program, not the CPU time.)

Computer users who routinely run the same programs would be the perfect candidates to evaluate a new computer. To evaluate a new system, these users would simply compare the execution time of their *workloads*—the mixture of programs

and operating system commands that users run on a computer. Few are in this happy situation, however. Most must rely on other methods to evaluate computers, and often other evaluators, hoping that these methods will predict performance for their usage of the new computer. One approach is benchmark programs, which are programs that many companies use to establish the relative performance of their computers.

Benchmarks

The best choice of benchmarks to measure performance is real applications, such as Google Translate mentioned in Section 1.1. Attempts at running programs that are much simpler than a real application have led to performance pitfalls. Examples include

- *Kernels*, which are small, key pieces of real applications.
- *Toy programs*, which are 100-line programs from beginning programming assignments, such as Quicksort.
- *Synthetic benchmarks*, which are fake programs invented to try to match the profile and behavior of real applications, such as Dhrystone.

All three are discredited today, usually because the compiler writer and architect can conspire to make the computer appear faster on these stand-in programs than on real applications. Regrettably for your authors—who dropped the fallacy about using synthetic benchmarks to characterize performance in the fourth edition of this book since we thought all computer architects agreed it was disreputable—the synthetic program Dhrystone is still the most widely quoted benchmark for embedded processors in 2017!

Another issue is the conditions under which the benchmarks are run. One way to improve the performance of a benchmark has been with benchmark-specific compiler flags; these flags often caused transformations that would be illegal on many programs or would slow down performance on others. To restrict this process and increase the significance of the results, benchmark developers typically require the vendor to use one compiler and one set of flags for all the programs in the same language (such as C++ or C). In addition to the question of compiler flags, another question is whether source code modifications are allowed. There are three different approaches to addressing this question:

1. No source code modifications are allowed.
2. Source code modifications are allowed but are essentially impossible. For example, database benchmarks rely on standard database programs that are tens of millions of lines of code. The database companies are highly unlikely to make changes to enhance the performance for one particular computer.
3. Source modifications are allowed, as long as the altered version produces the same output.

The key issue that benchmark designers face in deciding to allow modification of the source is whether such modifications will reflect real practice and provide useful insight to users, or whether these changes simply reduce the accuracy of the benchmarks as predictors of real performance. As we will see in Chapter 7, domain-specific architects often follow the third option when creating processors for well-defined tasks.

To overcome the danger of placing too many eggs in one basket, collections of benchmark applications, called *benchmark suites*, are a popular measure of performance of processors with a variety of applications. Of course, such collections are only as good as the constituent individual benchmarks. Nonetheless, a key advantage of such suites is that the weakness of any one benchmark is lessened by the presence of the other benchmarks. The goal of a benchmark suite is that it will characterize the real relative performance of two computers, particularly for programs not in the suite that customers are likely to run.

A cautionary example is the Electronic Design News Embedded Microprocessor Benchmark Consortium (or EEMBC, pronounced "embassy") benchmarks.

It is a set of 41 kernels used to predict performance of different embedded applications: automotive/industrial, consumer, networking, office automation, and telecommunications. EEMBC reports unmodified performance and "full fury" performance, where almost anything goes. Because these benchmarks use small kernels, and because of the reporting options, EEMBC does not have the reputation of being a good predictor of relative performance of different embedded computers in the field. This lack of success is why Dhrystone, which EEMBC was trying to replace, is sadly still used.

One of the most successful attempts to create standardized benchmark application suites has been the SPEC (Standard Performance Evaluation Corporation), which had its roots in efforts in the late 1980s to deliver better benchmarks for workstations. Just as the computer industry has evolved over time, so has the need for different benchmark suites, and there are now SPEC benchmarks to cover many application classes. All the SPEC benchmark suites and their reported results are found at http://www.spec.org.

Although we focus our discussion on the SPEC benchmarks in many of the following sections, many benchmarks have also been developed for PCs running the Windows operating system.

Desktop Benchmarks

Desktop benchmarks divide into two broad classes: processor-intensive benchmarks and graphics-intensive benchmarks, although many graphics benchmarks include intensive processor activity. SPEC originally created a benchmark set focusing on processor performance (initially called SPEC89), which has evolved into its sixth generation: SPEC CPU2017, which follows SPEC2006, SPEC2000, SPEC95 SPEC92, and SPEC89. SPEC CPU2017 consists of a set of 10 integer benchmarks (CINT2017) and 17 floating-point benchmarks (CFP2017). Figure 1.17 describes the current SPEC CPU benchmarks and their ancestry.

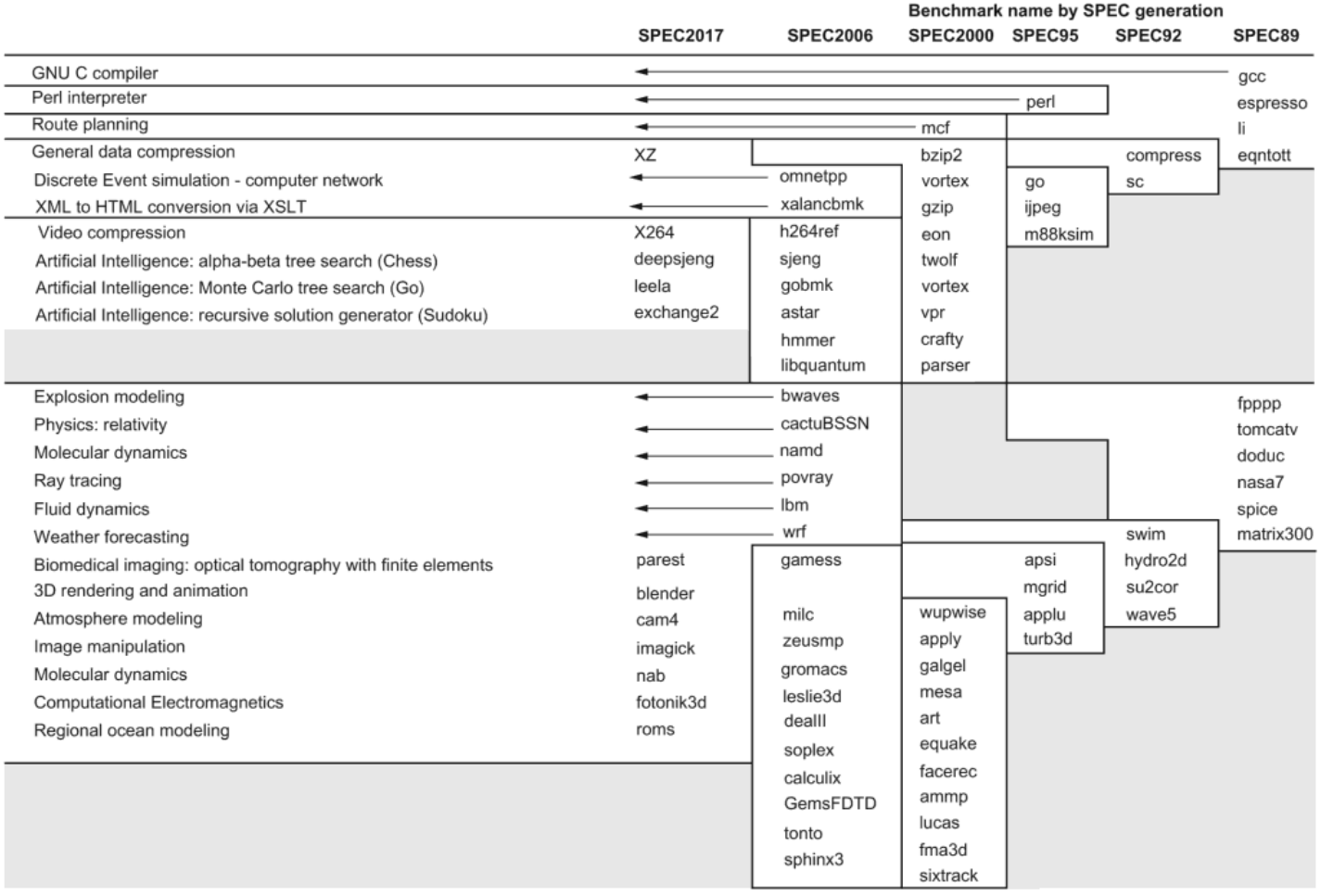


Figure 1.17 SPEC2017 programs and the evolution of the SPEC benchmarks over time, with integer programs above the line and floating-point programs below the line. Of the 10 SPEC2017 integer programs, 5 are written in C, 4 in C++., and 1 in Fortran. For the floating-point programs, the split is 3 in Fortran, 2 in C++, 2 in C, and 6 in mixed C, C++, and Fortran. The figure shows all 82 of the programs in the 1989, 1992, 1995, 2000, 2006, and 2017 releases. Gcc is the senior citizen of the group. Only 3 integer programs and 3 floating-point programs survived three or more generations. Although a few are carried over from generation to generation, the version of the program changes and either the input or the size of the benchmark is often expanded to increase its running time and to avoid perturbation in measurement or domination of the execution time by some factor other than CPU time. The benchmark descriptions on the left are for SPEC2017 only and do not apply to earlier versions. Programs in the same row from different generations of SPEC are generally not related; for example, fpppp is not a CFD code like bwaves.

SPEC benchmarks are real programs modified to be portable and to minimize the effect of I/O on performance. The integer benchmarks vary from part of a C compiler to a go program to a video compression. The floating-point benchmarks include molecular dynamics, ray tracing, and weather forecasting. The SPEC CPU suite is useful for processor benchmarking for both desktop systems and single-processor servers. We will see data on many of these programs throughout this book. However, these programs share little with modern programming languages and environments and the Google Translate application that Section 1.1 describes. Nearly half of them are written at least partially in Fortran! They are even statically linked instead of being dynamically linked like most real programs. Alas, the SPEC2017 applications themselves may be real, but they are not inspiring. It's not clear that SPECINT2017 and SPECFP2017 capture what is exciting about computing in the 21st century.

In Section 1.11, we describe pitfalls that have occurred in developing the SPEC CPUbenchmark suite, as well as the challenges in maintaining a useful and predictive benchmark suite.

SPEC CPU2017 is aimed at processor performance, but SPEC offers many other benchmarks. Figure 1.18 lists the 17 SPEC benchmarks that are active in 2017.

Server Benchmarks

Just as servers have multiple functions, so are there multiple types of benchmarks. The simplest benchmark is perhaps a processor throughput-oriented benchmark. SPEC CPU2017 uses the SPEC CPU benchmarks to construct a simple throughput benchmark where the processing rate of a multiprocessor can be measured by running multiple copies (usually as many as there are processors) of each SPEC CPU benchmark and converting the CPU time into a rate. This leads to a measurement called the SPECrate, and it is a measure of request-level parallelism from Section 1.2. To measure thread-level parallelism, SPEC offers what they call high-performance computing benchmarks around OpenMP and MPI as well as for accelerators such as GPUs (see Figure 1.18).

Other than SPECrate, most server applications and benchmarks have significant I/O activity arising from either storage or network traffic, including benchmarks for file server systems, for web servers, and for database and transaction-processing systems. SPEC offers both a file server benchmark (SPECSFS) and a Java server benchmark. (Appendix D discusses some file and I/O system benchmarks in detail.) SPECvirt_Sc2013 evaluates end-to-end performance of virtualized data center servers. Another SPEC benchmark measures power, which we examine in Section 1.10.

Transaction-processing (TP) benchmarks measure the ability of a system to handle transactions that consist of database accesses and updates. Airline reservation systems and bank ATM systems are typical simple examples of TP; more sophisticated TP systems involve complex databases and decision-making.

Category	Name	Measures performance of
Cloud	Cloud_IaaS 2016	Cloud using NoSQL database transaction and K-Means clustering using map/reduce
CPU	CPU2017	Compute-intensive integer and floating-point workloads
Graphics and workstation performance	SPECviewperf® 12	3D graphics in systems running OpenGL and Direct X
	SPECwpc V2.0	Workstations running professional apps under the Windows OS
	SPECapcSM for 3ds Max 2015™	3D graphics running the proprietary Autodesk 3ds Max 2015 app
	SPECapcSM for Maya® 2012	3D graphics running the proprietary Autodesk 3ds Max 2012 app
	SPECapcSM for PTC Creo 3.0	3D graphics running the proprietary PTC Creo 3.0 app
	SPECapcSM for Siemens NX 9.0 and 10.0	3D graphics running the proprietary Siemens NX 9.0 or 10.0 app
	SPECapcSM for SolidWorks 2015	3D graphics of systems running the proprietary SolidWorks 2015 CAD/CAM app
High performance computing	ACCEL	Accelerator and host CPU running parallel applications using OpenCL and OpenACC
	MPI2007	MPI-parallel, floating-point, compute-intensive programs running on clusters and SMPs
	OMP2012	Parallel apps running OpenMP
Java client/server	SPECjbb2015	Java servers
Power	SPECpower_ssj2008	Power of volume server class computers running SPECjbb2015
Solution File Server (SFS)	SFS2014	File server throughput and response time
	SPECsfs2008	File servers utilizing the NFSv3 and CIFS protocols
Virtualization	SPECvirt_sc2013	Datacenter servers used in virtualized server consolidation

Figure 1.18 Active benchmarks from SPEC as of 2017.

In the mid-1980s, a group of concerned engineers formed the vendor-independent Transaction Processing Council (TPC) to try to create realistic and fair benchmarks for TP. The TPC benchmarks are described at http://www.tpc.org.

The first TPC benchmark, TPC-A, was published in 1985 and has since been replaced and enhanced by several different benchmarks. TPC-C, initially created in 1992, simulates a complex query environment. TPC-H models ad hoc decision support—the queries are unrelated and knowledge of past queries cannot be used to optimize future queries. The TPC-DI benchmark, a new data integration (DI) task also known as ETL, is an important part of data warehousing. TPC-E is an online transaction processing (OLTP) workload that simulates a brokerage firm's customer accounts.

Recognizing the controversy between traditional relational databases and "No SQL" storage solutions, TPCx-HS measures systems using the Hadoop file system running MapReduce programs, and TPC-DS measures a decision support system that uses either a relational database or a Hadoop-based system. TPC-VMS and TPCx-V measure database performance for virtualized systems, and TPC-Energy adds energy metrics to all the existing TPC benchmarks.

All the TPC benchmarks measure performance in transactions per second. In addition, they include a response time requirement so that throughput performance is measured only when the response time limit is met. To model real-world systems, higher transaction rates are also associated with larger systems, in terms of both users and the database to which the transactions are applied. Finally, the system cost for a benchmark system must be included as well to allow accurate comparisons of cost-performance. TPC modified its pricing policy so that there is a single specification for all the TPC benchmarks and to allow verification of the prices that TPC publishes.

Reporting Performance Results

The guiding principle of reporting performance measurements should be *reproducibility*—list everything another experimenter would need to duplicate the results. A SPEC benchmark report requires an extensive description of the computer and the compiler flags, as well as the publication of both the baseline and the optimized results. In addition to hardware, software, and baseline tuning parameter descriptions, a SPEC report contains the actual performance times, shown both in tabular form and as a graph. A TPC benchmark report is even more complete, because it must include results of a benchmarking audit and cost information. These reports are excellent sources for finding the real costs of computing systems, since manufacturers compete on high performance and cost-performance.

Summarizing Performance Results

In practical computer design, one must evaluate myriad design choices for their relative quantitative benefits across a suite of benchmarks believed to be relevant. Likewise, consumers trying to choose a computer will rely on performance measurements from benchmarks, which ideally are similar to the users' applications. In both cases, it is useful to have measurements for a suite of benchmarks so that the performance of important applications is similar to that of one or more benchmarks in the suite and so that variability in performance can be understood. In the best case, the suite resembles a statistically valid sample of the application space, but such a sample requires more benchmarks than are typically found in most suites and requires a randomized sampling, which essentially no benchmark suite uses.

Once we have chosen to measure performance with a benchmark suite, we want to be able to summarize the performance results of the suite in a unique number. A simple approach to computing a summary result would be to compare the arithmetic means of the execution times of the programs in the suite. An alternative would be to add a weighting factor to each benchmark and use the weighted arithmetic mean as the single number to summarize performance. One approach is to use weights that make all programs execute an equal time on some reference computer, but this biases the results toward the performance characteristics of the reference computer.

Rather than pick weights, we could normalize execution times to a reference computer by dividing the time on the reference computer by the time on the computer being rated, yielding a ratio proportional to performance. SPEC uses this approach, calling the ratio the SPECRatio. It has a particularly useful property that matches the way we benchmark computer performance throughout this text—namely, comparing performance ratios. For example, suppose that the SPECRatio of computer A on a benchmark is 1.25 times as fast as computer B; then we know

$$1.25 = \frac{\text{SPECRatio}_A}{\text{SPECRatio}_B} = \frac{\dfrac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_A}}{\dfrac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_B}} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{\text{Performance}_A}{\text{Performance}_B}$$

Notice that the execution times on the reference computer drop out and the choice of the reference computer is irrelevant when the comparisons are made as a ratio, which is the approach we consistently use. Figure 1.19 gives an example.

Because a SPECRatio is a ratio rather than an absolute execution time, the mean must be computed using the *geometric* mean. (Because SPECRatios have no units, comparing SPECRatios arithmetically is meaningless.) The formula is

$$\text{Geometric mean} = \sqrt[n]{\prod_{i=1}^{n} sample_i}$$

In the case of SPEC, $sample_i$ is the SPECRatio for program i. Using the geometric mean ensures two important properties:

1. The geometric mean of the ratios is the same as the ratio of the geometric means.
2. The ratio of the geometric means is equal to the geometric mean of the performance ratios, which implies that the choice of the reference computer is irrelevant.

Therefore the motivations to use the geometric mean are substantial, especially when we use performance ratios to make comparisons.

Example Show that the ratio of the geometric means is equal to the geometric mean of the performance ratios and that the reference computer of SPECRatio does not matter.

Answer Assume two computers A and B and a set of SPECRatios for each.

$$\frac{\text{Geometric mean}_A}{\text{Geometric mean}_B} = \frac{\sqrt[n]{\prod_{i=1}^{n} \text{SPECRatio A}_i}}{\sqrt[n]{\prod_{i=1}^{n} \text{SPECRatio B}_i}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{SPECRatio A}_i}{\text{SPECRatio B}_i}}$$

$$= \sqrt[n]{\prod_{i=1}^{n} \frac{\dfrac{\text{Execution time}_{\text{reference}_i}}{\text{Execution time}_{A_i}}}{\dfrac{\text{Execution time}_{\text{reference}_i}}{\text{Execution time}_{B_i}}}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{Execution time}_{B_i}}{\text{Execution time}_{A_i}}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{Performance}_{A_i}}{\text{Performance}_{B_i}}}$$

That is, the ratio of the geometric means of the SPECRatios of A and B is the geometric mean of the performance ratios of A to B of all the benchmarks in the suite. Figure 1.19 demonstrates this validity using examples from SPEC.

Benchmarks	Sun Ultra Enterprise 2 time (seconds)	AMD A10-6800K time (seconds)	SPEC 2006Cint ratio	Intel Xeon E5-2690 time (seconds)	SPEC 2006Cint ratio	AMD/Intel times (seconds)	Intel/AMD SPEC ratios
perlbench	9770	401	24.36	261	37.43	1.54	1.54
bzip2	9650	505	19.11	422	22.87	1.20	1.20
gcc	8050	490	16.43	227	35.46	2.16	2.16
mcf	9120	249	36.63	153	59.61	1.63	1.63
gobmk	10,490	418	25.10	382	27.46	1.09	1.09
hmmer	9330	182	51.26	120	77.75	1.52	1.52
sjeng	12,100	517	23.40	383	31.59	1.35	1.35
libquantum	20,720	84	246.08	3	7295.77	29.65	29.65
h264ref	22,130	611	36.22	425	52.07	1.44	1.44
omnetpp	6250	313	19.97	153	40.85	2.05	2.05
astar	7020	303	23.17	209	33.59	1.45	1.45
xalancbmk	6900	215	32.09	98	70.41	2.19	2.19
Geometric mean			31.91		63.72	2.00	2.00

Figure 1.19 SPEC2006Cint execution times (in seconds) for the Sun Ultra 5—the reference computer of SPEC2006—and execution times and SPECRatios for the AMD A10 and Intel Xeon E5-2690. The final two columns show the ratios of execution times and SPEC ratios. This figure demonstrates the irrelevance of the reference computer in relative performance. The ratio of the execution times is identical to the ratio of the SPEC ratios, and the ratio of the geometric means (63.7231.91/20.86 = 2.00) is identical to the geometric mean of the ratios (2.00). Section 1.11 discusses libquantum, whose performance is orders of magnitude higher than the other SPEC benchmarks.

1.9 Quantitative Principles of Computer Design

Now that we have seen how to define, measure, and summarize performance, cost, dependability, energy, and power, we can explore guidelines and principles that are useful in the design and analysis of computers. This section introduces important observations about design, as well as two equations to evaluate alternatives.

Take Advantage of Parallelism

Using parallelism is one of the most important methods for improving performance. Every chapter in this book has an example of how performance is enhanced through the exploitation of parallelism. We give three brief examples here, which are expounded on in later chapters.

Our first example is the use of parallelism at the system level. To improve the throughput performance on a typical server benchmark, such as SPECSFS or TPC-C, multiple processors and multiple storage devices can be used. The workload of handling requests can then be spread among the processors and storage devices, resulting in improved throughput. Being able to expand memory and the number of processors and storage devices is called *scalability*, and it is a valuable asset for servers. Spreading of data across many storage devices for parallel reads and writes enables data-level parallelism. SPECSFS also relies on request-level parallelism to use many processors, whereas TPC-C uses thread-level parallelism for faster processing of database queries.

At the level of an individual processor, taking advantage of parallelism among instructions is critical to achieving high performance. One of the simplest ways to do this is through pipelining. (Pipelining is explained in more detail in Appendix C and is a major focus of Chapter 3.) The basic idea behind pipelining is to overlap instruction execution to reduce the total time to complete an instruction sequence. A key insight into pipelining is that not every instruction depends on its immediate predecessor, so executing the instructions completely or partially in parallel may be possible. Pipelining is the best-known example of ILP.

Parallelism can also be exploited at the level of detailed digital design. For example, set-associative caches use multiple banks of memory that are typically searched in parallel to find a desired item. Arithmetic-logical units use carry-lookahead, which uses parallelism to speed the process of computing sums from linear to logarithmic in the number of bits per operand. These are more examples of *data-level parallelism*.

Principle of Locality

Important fundamental observations have come from properties of programs. The most important program property that we regularly exploit is the *principle of locality*: programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that we can predict with reasonable

accuracy what instructions and data a program will use in the near future based on its accesses in the recent past. The principle of locality also applies to data accesses, though not as strongly as to code accesses.

Two different types of locality have been observed. *Temporal locality* states that recently accessed items are likely to be accessed soon. *Spatial locality* says that items whose addresses are near one another tend to be referenced close together in time. We will see these principles applied in Chapter 2.

Focus on the Common Case

Perhaps the most important and pervasive principle of computer design is to focus on the common case: in making a design trade-off, favor the frequent case over the infrequent case. This principle applies when determining how to spend resources, because the impact of the improvement is higher if the occurrence is commonplace.

Focusing on the common case works for energy as well as for resource allocation and performance. The instruction fetch and decode unit of a processor may be used much more frequently than a multiplier, so optimize it first. It works on dependability as well. If a database server has 50 storage devices for every processor, storage dependability will dominate system dependability.

In addition, the common case is often simpler and can be done faster than the infrequent case. For example, when adding two numbers in the processor, we can expect overflow to be a rare circumstance and can therefore improve performance by optimizing the more common case of no overflow. This emphasis may slow down the case when overflow occurs, but if that is rare, then overall performance will be improved by optimizing for the normal case.

We will see many cases of this principle throughout this text. In applying this simple principle, we have to decide what the frequent case is and how much performance can be improved by making that case faster. A fundamental law, called *Amdahl's Law*, can be used to quantify this principle.

Amdahl's Law

The performance gain that can be obtained by improving some portion of a computer can be calculated using Amdahl's Law. Amdahl's Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

Amdahl's Law defines the *speedup* that can be gained by using a particular feature. What is speedup? Suppose that we can make an enhancement to a computer that will improve performance when it is used. Speedup is the ratio

$$\text{Speedup} = \frac{\text{Performance for entire task using the enhancement when possible}}{\text{Performance for entire task without using the enhancement}}$$

Alternatively,

$$\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$

Speedup tells us how much faster a task will run using the computer with the enhancement contrary to the original computer.

Amdahl's Law gives us a quick way to find the speedup from some enhancement, which depends on two factors:

1. *The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement*—For example, if 40 seconds of the execution time of a program that takes 100 seconds in total can use an enhancement, the fraction is 40/100. This value, which we call $\text{Fraction}_{\text{enhanced}}$, is always less than or equal to 1.
2. *The improvement gained by the enhanced execution mode, that is, how much faster the task would run if the enhanced mode were used for the entire program*—This value is the time of the original mode over the time of the enhanced mode. If the enhanced mode takes, say, 4 seconds for a portion of the program, while it is 40 seconds in the original mode, the improvement is 40/4 or 10. We call this value, which is always greater than 1, $\text{Speedup}_{\text{enhanced}}$.

The execution time using the original computer with the enhanced mode will be the time spent using the unenhanced portion of the computer plus the time spent using the enhancement:

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

The overall speedup is the ratio of the execution times:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Example Suppose that we want to enhance the processor used for web serving. The new processor is 10 times faster on computation in the web serving application than the old processor. Assuming that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

Answer $\text{Fraction}_{\text{enhanced}} = 0.4$; $\text{Speedup}_{\text{enhanced}} = 10$; $\text{Speedup}_{\text{overall}} = \dfrac{1}{0.6 + \frac{0.4}{10}} = \dfrac{1}{0.64} \approx 1.56$

Amdahl's Law expresses the law of diminishing returns: The incremental improvement in speedup gained by an improvement of just a portion of the computation diminishes as improvements are added. An important corollary of Amdahl's Law is that if an enhancement is usable only for a fraction of a task, then we can't speed up the task by more than the reciprocal of 1 minus that fraction.

A common mistake in applying Amdahl's Law is to confuse "fraction of time converted *to use an enhancement*" and "fraction of time *after enhancement is in use*." If, instead of measuring the time that we *could use* the enhancement in a computation, we measure the time *after* the enhancement is in use, the results will be incorrect!

Amdahl's Law can serve as a guide to how much an enhancement will improve performance and how to distribute resources to improve cost-performance. The goal, clearly, is to spend resources proportional to where time is spent. Amdahl's Law is particularly useful for comparing the overall system performance of two alternatives, but it can also be applied to compare two processor design alternatives, as the following example shows.

Example A common transformation required in graphics processors is square root. Implementations of floating-point (FP) square root vary significantly in performance, especially among processors designed for graphics. Suppose FP square root (FSQRT) is responsible for 20% of the execution time of a critical graphics benchmark. One proposal is to enhance the FSQRT hardware and speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions in the graphics processor run faster by a factor of 1.6; FP instructions are responsible for half of the execution time for the application. The design team believes that they can make all FP instructions run 1.6 times faster with the same effort as required for the fast square root. Compare these two design alternatives.

Answer We can compare these two alternatives by comparing the speedups:

$$\text{Speedup}_{\text{FSQRT}} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

Improving the performance of the FP operations overall is slightly better because of the higher frequency.

Amdahl's Law is applicable beyond performance. Let's redo the reliability example from page 39 after improving the reliability of the power supply via redundancy from 200,000-hour to 830,000,000-hour MTTF, or 4150× better.

Example The calculation of the failure rates of the disk subsystem was

$$\text{Failure rate}_{\text{system}} = 10 \times \frac{1}{1{,}000{,}000} + \frac{1}{500{,}000} + \frac{1}{200{,}000} + \frac{1}{200{,}000} + \frac{1}{1{,}000{,}000}$$

$$= \frac{10+2+5+5+1}{1{,}000{,}000 \text{ hours}} = \frac{23}{1{,}000{,}000 \text{ hours}}$$

Therefore the fraction of the failure rate that could be improved is 5 per million hours out of 23 for the whole system, or 0.22.

Answer The reliability improvement would be

$$\text{Improvement}_{\text{power supply pair}} = \frac{1}{(1-0.22)+\frac{0.22}{4150}} = \frac{1}{0.78} = 1.28$$

Despite an impressive 4150 × improvement in reliability of one module, from the system's perspective, the change has a measurable but small benefit.

In the preceding examples, we needed the fraction consumed by the new and improved version; often it is difficult to measure these times directly. In the next section, we will see another way of doing such comparisons based on the use of an equation that decomposes the CPU execution time into three separate components. If we know how an alternative affects these three components, we can determine its overall performance. Furthermore, it is often possible to build simulators that measure these components before the hardware is actually designed.

The Processor Performance Equation

Essentially all computers are constructed using a clock running at a constant rate. These discrete time events are called *clock periods*, *clocks*, *cycles*, or *clock cycles*. Computer designers refer to the time of a clock period by its duration (e.g., 1 ns) or by its rate (e.g., 1 GHz). CPU time for a program can then be expressed two ways:

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

or

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

In addition to the number of clock cycles needed to execute a program, we can also count the number of instructions executed—the *instruction path length* or *instruction count* (IC). If we know the number of clock cycles and the instruction count, we can calculate the average number of *clock cycles per instruction* (CPI). Because it is easier to work with, and because we will deal with simple processors in this chapter, we use CPI. Designers sometimes also use *instructions per clock* (IPC), which is the inverse of CPI.

CPI is computed as

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

This processor figure of merit provides insight into different styles of instruction sets and implementations, and we will use it extensively in the next four chapters.

By transposing the instruction count in the preceding formula, clock cycles can be defined as IC × CPI. This allows us to use CPI in the execution time formula:

$$\text{CPU time} = \text{Instruction count} \times \text{Cycles per instruction} \times \text{Clock cycle time}$$

Expanding the first formula into the units of measurement shows how the pieces fit together:

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

As this formula demonstrates, processor performance is dependent upon three characteristics: clock cycle (or rate), clock cycles per instruction, and instruction count. Furthermore, CPU time is *equally* dependent on these three characteristics; for example, a 10% improvement in any one of them leads to a 10% improvement in CPU time.

Unfortunately, it is difficult to change one parameter in complete isolation from others because the basic technologies involved in changing each characteristic are interdependent:

- *Clock cycle time*—Hardware technology and organization
- *CPI*—Organization and instruction set architecture
- *Instruction count*—Instruction set architecture and compiler technology

Luckily, many potential performance improvement techniques primarily enhance one component of processor performance with small or predictable impacts on the other two.

In designing the processor, sometimes it is useful to calculate the number of total processor clock cycles as

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

where IC_i represents the number of times instruction i is executed in a program and CPI_i represents the average number of clocks per instruction for instruction i. This form can be used to express CPU time as

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

and overall CPI as

$$\text{CPI} = \frac{\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i}{\text{Instruction count}} = \sum_{i=1}^{n} \frac{\text{IC}_i}{\text{Instruction count}} \times \text{CPI}_i$$

The latter form of the CPI calculation uses each individual CPI_i and the fraction of occurrences of that instruction in a program (i.e., $\text{IC}_i \div$ Instruction count). Because it must include pipeline effects, cache misses, and any other memory system

inefficiencies, CPI_i should be measured and not just calculated from a table in the back of a reference manual.

Consider our performance example on page 52, here modified to use measurements of the frequency of the instructions and of the instruction CPI values, which, in practice, are obtained by simulation or by hardware instrumentation.

Example Suppose we made the following measurements:

Frequency of FP operations = 25%

Average CPI of FP operations = 4.0

Average CPI of other instructions = 1.33

Frequency of FSQRT = 2%

CPI of FSQRT = 20

Assume that the two design alternatives are to decrease the CPI of FSQRT to 2 or to decrease the average CPI of all FP operations to 2.5. Compare these two design alternatives using the processor performance equation.

Answer First, observe that only the CPI changes; the clock rate and instruction count remain identical. We start by finding the original CPI with neither enhancement:

$$\begin{aligned}\text{CPI}_{\text{original}} &= \sum_{i=1}^{n} \text{CPI}_i \times \left(\frac{\text{IC}_i}{\text{Instruction count}}\right)\\ &= (4 \times 25\%) + (1.33 \times 75\%) = 2.0\end{aligned}$$

We can compute the CPI for the enhanced FSQRT by subtracting the cycles saved from the original CPI:

$$\begin{aligned}\text{CPI}_{\text{with new FPSQR}} &= \text{CPI}_{\text{original}} - 2\% \times \left(\text{CPI}_{\text{old FPSQR}} - \text{CPI}_{\text{of new FPSQR only}}\right)\\ &= 2.0 - 2\% \times (20 - 2) = 1.64\end{aligned}$$

We can compute the CPI for the enhancement of all FP instructions the same way or by summing the FP and non-FP CPIs. Using the latter gives us

$$\text{CPI}_{\text{new FP}} = (75\% \times 1.33) + (25\% \times 2.5) = 1.625$$

Since the CPI of the overall FP enhancement is slightly lower, its performance will be marginally better. Specifically, the speedup for the overall FP enhancement is

$$\begin{aligned}\text{Speedup}_{\text{new FP}} &= \frac{\text{CPU time}_{\text{original}}}{\text{CPU time}_{\text{new FP}}} = \frac{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{original}}}{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{new FP}}}\\ &= \frac{\text{CPI}_{\text{original}}}{\text{CPI}_{\text{new FP}}} = \frac{2.00}{1.625} = 1.23\end{aligned}$$

Happily, we obtained this same speedup using Amdahl's Law on page 51.

It is often possible to measure the constituent parts of the processor performance equation. Such isolated measurements are a key advantage of using the processor performance equation versus Amdahl's Law in the previous example. In particular, it may be difficult to measure things such as the fraction of execution time for which a set of instructions is responsible. In practice, this would probably be computed by summing the product of the instruction count and the CPI for each of the instructions in the set. Since the starting point is often individual instruction count and CPI measurements, the processor performance equation is incredibly useful.

To use the processor performance equation as a design tool, we need to be able to measure the various factors. For an existing processor, it is easy to obtain the execution time by measurement, and we know the default clock speed. The challenge lies in discovering the instruction count or the CPI. Most processors include counters for both instructions executed and clock cycles. By periodically monitoring these counters, it is also possible to attach execution time and instruction count to segments of the code, which can be helpful to programmers trying to understand and tune the performance of an application. Often designers or programmers will want to understand performance at a more fine-grained level than what is available from the hardware counters. For example, they may want to know why the CPI is what it is. In such cases, the simulation techniques used are like those for processors that are being designed.

Techniques that help with energy efficiency, such as dynamic voltage frequency scaling and overclocking (see Section 1.5), make this equation harder to use, because the clock speed may vary while we measure the program. A simple approach is to turn off those features to make the results reproducible. Fortunately, as performance and energy efficiency are often highly correlated—taking less time to run a program generally saves energy—it's probably safe to consider performance without worrying about the impact of DVFS or overclocking on the results.

1.10 Putting It All Together: Performance, Price, and Power

In the "Putting It All Together" sections that appear near the end of every chapter, we provide real examples that use the principles in that chapter. In this section, we look at measures of performance and power-performance in small servers using the SPECpower benchmark.

Figure 1.20 shows the three multiprocessor servers we are evaluating along with their price. To keep the price comparison fair, all are Dell PowerEdge servers. The first is the PowerEdge R710, which is based on the Intel Xeon ×85670 microprocessor with a clock rate of 2.93 GHz. Unlike the Intel Core i7-6700 in Chapters 2–5, which has 20 cores and a 40 MB L3 cache, this Intel chip has 22 cores and a 55 MB L3 cache, although the cores themselves are identical. We selected a two-socket system—so 44 cores total—with 128 GB of ECC-protected 2400 MHz DDR4 DRAM. The next server is the PowerEdge C630, with the same processor, number of sockets, and DRAM. The main difference is a smaller rack-mountable package: "2U" high (3.5 inches) for the 730 versus "1U" (1.75 inches) for the 630.

	System 1		System 2		System 3	
Component		**Cost (% Cost)**		**Cost (% Cost)**		**Cost (% Cost)**
Base server	PowerEdge R710	$653 (7%)	PowerEdge R815	$1437 (15%)	PowerEdge R815	$1437 (11%)
Power supply	570 W		1100 W		1100 W	
Processor	Xeon X5670	$3738 (40%)	Opteron 6174	$2679 (29%)	Opteron 6174	$5358 (42%)
Clock rate	2.93 GHz		2.20 GHz		2.20 GHz	
Total cores	12		24		48	
Sockets	2		2		4	
Cores/socket	6		12		12	
DRAM	12 GB	$484 (5%)	16 GB	$693 (7%)	32 GB	$1386 (11%)
Ethernet Inter.	Dual 1-Gbit	$199 (2%)	Dual 1-Gbit	$199 (2%)	Dual 1-Gbit	$199 (2%)
Disk	50 GB SSD	$1279 (14%)	50 GB SSD	$1279 (14%)	50 GB SSD	$1279 (10%)
Windows OS		$2999 (32%)		$2999 (33%)		$2999 (24%)
Total		$9352 (100%)		$9286 (100%)		$12,658 (100%)
Max ssj_ops	910,978		926,676		1,840,450	
Max ssj_ops/$	97		100		145	

Figure 1.20 Three Dell PowerEdge servers being measured and their prices as of July 2016. We calculated the cost of the processors by subtracting the cost of a second processor. Similarly, we calculated the overall cost of memory by seeing what the cost of extra memory was. Hence the base cost of the server is adjusted by removing the estimated cost of the default processor and memory. Chapter 5 describes how these multisocket systems are connected together, and Chapter 6 describes how clusters are connected together.

The third server is a cluster of 16 of the PowerEdge 630 s that is connected together with a 1 Gbit/s Ethernet switch. All are running the Oracle Java HotSpot version 1.7 Java Virtual Machine (JVM) and the Microsoft Windows Server 2012 R2 Datacenter version 6.3 operating system.

Note that because of the forces of benchmarking (see Section 1.11), these are unusually configured servers. The systems in Figure 1.20 have little memory relative to the amount of computation, and just a tiny 120 GB solid-state disk. It is inexpensive to add cores if you don't need to add commensurate increases in memory and storage!

Rather than run statically linked C programs of SPEC CPU, SPECpower uses a more modern software stack written in Java. It is based on SPECjbb, and it represents the server side of business applications, with performance measured as the number of transactions per second, called *ssj_ops* for *server side Java operations per second*. It exercises not only the processor of the server, as does SPEC CPU, but also the caches, memory system, and even the multiprocessor interconnection system. In addition, it exercises the JVM, including the JIT runtime compiler and garbage collector, as well as portions of the underlying operating system.

As the last two rows of Figure 1.20 show, the performance winner is the cluster of 16 R630s, which is hardly a surprise since it is by far the most expensive. The price-performance winner is the PowerEdge R630, but it barely beats the cluster at 213 versus 211 ssj-ops/$. Amazingly, the 16 node cluster is within 1% of the same price-performances of a single node despite being 16 times as large.

While most benchmarks (and most computer architects) care only about performance of systems at peak load, computers rarely run at peak load. Indeed, Figure 6.2 in Chapter 6 shows the results of measuring the utilization of tens of thousands of servers over 6 months at Google, and less than 1% operate at an average utilization of 100%. The majority have an average utilization of between 10% and 50%. Thus the SPECpower benchmark captures power as the target workload varies from its peak in 10% intervals all the way to 0%, which is called Active Idle.

Figure 1.21 plots the ssj_ops (SSJ operations/second) per watt and the average power as the target load varies from 100% to 0%. The Intel R730 always has the lowest power and the single node R630 has the best ssj_ops per watt across each target workload level. Since watts = joules/second, this metric is proportional to SSJ operations per joule:

$$\frac{\text{ssj\_operations/second}}{\text{Watt}} = \frac{\text{ssj\_operations/second}}{\text{Joule/second}} = \frac{\text{ssj\_operations}}{\text{Joule}}$$

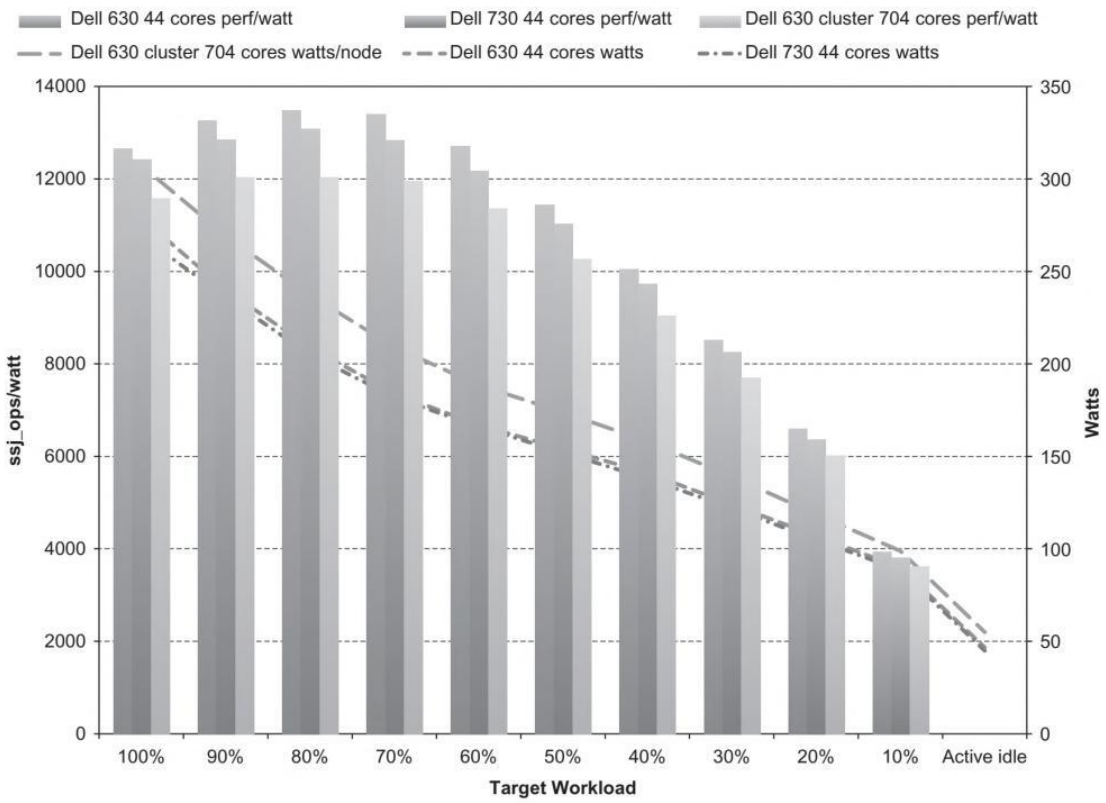


Figure 1.21 Power-performance of the three servers in Figure 1.20. Ssj_ops/watt values are on the left axis, with the three columns associated with it, and watts are on the right axis, with the three lines associated with it. The horizontal axis shows the target workload, as it varies from 100% to Active Idle. The single node R630 has the best ssj_ops/watt at each workload level, but R730 consumes the lowest power at each level.

To calculate a single number to use to compare the power efficiency of systems, SPECpower uses

$$\text{Overall ssj\_ops/watt} = \frac{\sum \text{ssj\_ops}}{\sum \text{power}}$$

The overall ssj_ops/watt of the three servers is 10,802 for the R730, 11,157 for the R630, and 10,062 for the cluster of 16 R630s. Therefore the single node R630 has the best power-performance. Dividing by the price of the servers, the ssj_ops/watt/$1,000 is 879 for the R730, 899 for the R630, and 789 (per node) for the 16-node cluster of R630s. Thus, after adding power, the single-node R630 is still in first place in performance/price, but now the single-node R730 is significantly more efficient than the 16-node cluster.

1.11 Fallacies and Pitfalls

The purpose of this section, which will be found in every chapter, is to explain some commonly held misbeliefs or misconceptions that you should avoid. We call such misbeliefs *fallacies.* When discussing a fallacy, we try to give a counterexample. We also discuss *pitfalls*—easily made mistakes. Often pitfalls are generalizations of principles that are true in a limited context. The purpose of these sections is to help you avoid making these errors in computers that you design.

Pitfall *All exponential laws must come to an end.*

The first to go was Dennard scaling. Dennard's 1974 observation was that power density was constant as transistors got smaller. If a transistor's linear region shrank by a factor 2, then both the current and voltage were also reduced by a factor of 2, and so the power it used fell by 4. Thus chips could be designed to operate faster and still use less power. Dennard scaling ended 30 years after it was observed, not because transistors didn't continue to get smaller but because integrated circuit dependability limited how far current and voltage could drop. The threshold voltage was driven so low that static power became a significant fraction of overall power.

The next deceleration was hard disk drives. Although there was no law for disks, in the past 30 years the maximum areal density of hard drives—which determines disk capacity—improved by 30%–100% per year. In more recent years, it has been less than 5% per year. Increasing density per drive has come primarily from adding more platters to a hard disk drive.

Next up was the venerable Moore's Law. It's been a while since the number of transistors per chip doubled every one to two years. For example, the DRAM chip introduced in 2014 contained 8B transistors, and we won't have a 16B transistor DRAM chip in mass production until 2019, but Moore's Law predicts a 64B transistor DRAM chip.

Moreover, the actual end of scaling of the planar logic transistor was even predicted to end by 2021. Figure 1.22 shows the predictions of the physical gate length

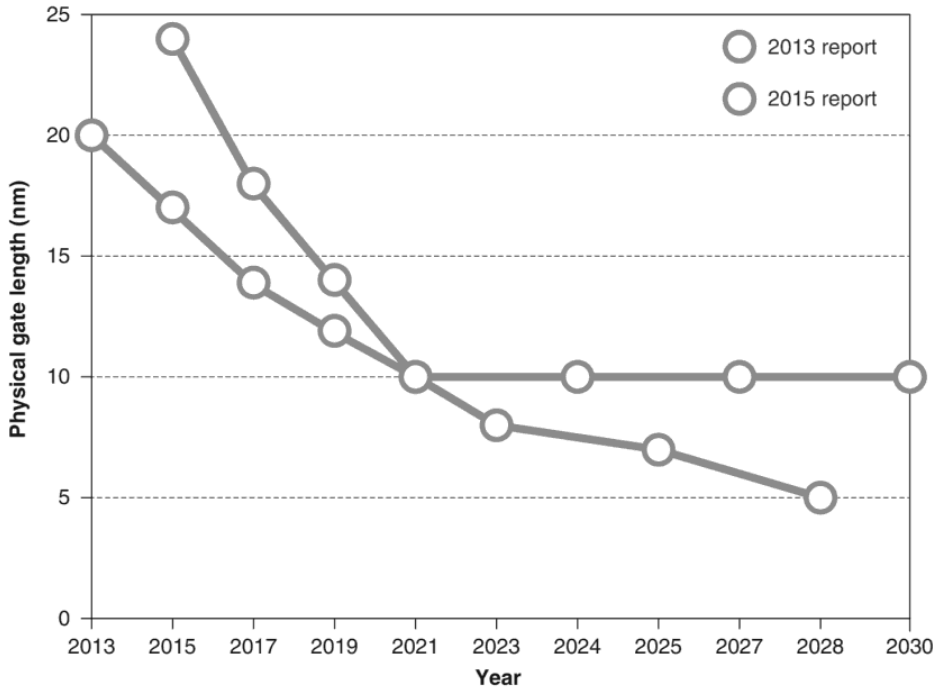


Figure 1.22 Predictions of logic transistor dimensions from two editions of the ITRS report. These reports started in 2001, but 2015 will be the last edition, as the group has disbanded because of waning interest. The only companies that can produce state-of-the-art logic chips today are GlobalFoundaries, Intel, Samsung, and TSMC, whereas there were 19 when the first ITRS report was released. With only four companies left, sharing of plans was too hard to sustain. From IEEE Spectrum, July 2016, "Transistors will stop shrinking in 2021, Moore's Law Roadmap Predicts," by Rachel Courtland.

of the logic transistor from two editions of the International Technology Roadmap for Semiconductors (ITRS). Unlike the 2013 report that projected gate lengths to reach 5 nm by 2028, the 2015 report projects the length stopping at 10 nm by 2021. Density improvements thereafter would have to come from ways other than shrinking the dimensions of transistors. It's not as dire as the ITRS suggests, as companies like Intel and TSMC have plans to shrink to 3 nm gate lengths, but the rate of change is decreasing.

Figure 1.23 shows the changes in increases in bandwidth over time for microprocessors and DRAM—which are affected by the end of Dennard scaling and Moore's Law—as well as for disks. The slowing of technology improvements is apparent in the dropping curves. The continued networking improvement is due to advances in fiber optics and a planned change in pulse amplitude modulation (PAM-4) allowing two-bit encoding so as to transmit information at 400 Gbit/s.

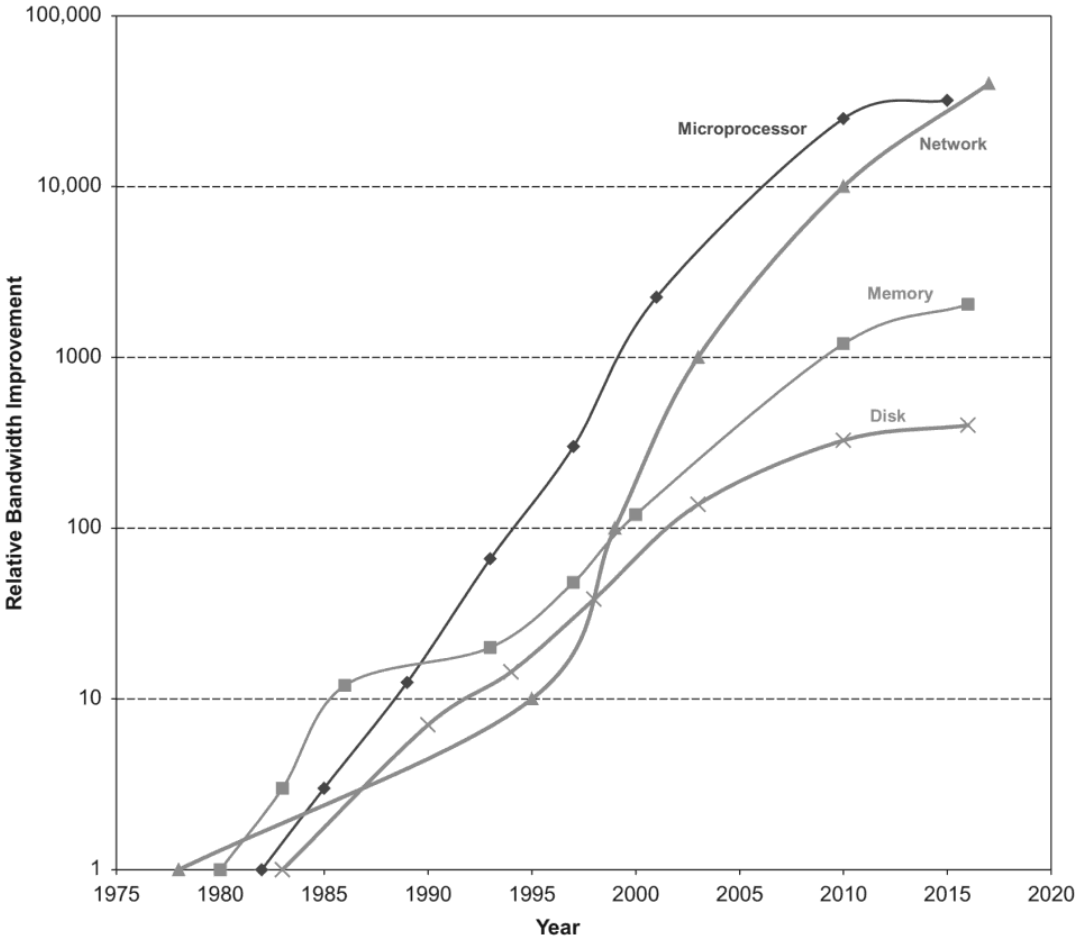


Figure 1.23 Relative bandwidth for microprocessors, networks, memory, and disks over time, based on data in Figure 1.10.

Fallacy *Multiprocessors are a silver bullet.*

The switch to multiple processors per chip around 2005 did not come from some breakthrough that dramatically simplified parallel programming or made it easy to build multicore computers. The change occurred because there was no other option due to the ILP walls and power walls. Multiple processors per chip do not guarantee lower power; it's certainly feasible to design a multicore chip that uses more power. The potential is just that it's possible to continue to improve performance by replacing a high-clock-rate, inefficient core with several lower-clock-rate, efficient cores. As technology to shrink transistors improves, it can shrink both capacitance and the supply voltage a bit so that we can get a modest increase in the

number of cores per generation. For example, for the past few years, Intel has been adding two cores per generation in their higher-end chips.

As we will see in Chapters 4 and 5, performance is now a programmer's burden. The programmers' La-Z-Boy era of relying on a hardware designer to make their programs go faster without lifting a finger is officially over. If programmers want their programs to go faster with each generation, they must make their programs more parallel.

The popular version of Moore's law—increasing performance with each generation of technology—is now up to programmers.

Pitfall *Falling prey to Amdahl's heartbreaking law.*

Virtually every practicing computer architect knows Amdahl's Law. Despite this, we almost all occasionally expend tremendous effort optimizing some feature before we measure its usage. Only when the overall speedup is disappointing do we recall that we should have measured first before we spent so much effort enhancing it!

Pitfall *A single point of failure.*

The calculations of reliability improvement using Amdahl's Law on page 53 show that dependability is no stronger than the weakest link in a chain. No matter how much more dependable we make the power supplies, as we did in our example, the single fan will limit the reliability of the disk subsystem. This Amdahl's Law observation led to a rule of thumb for fault-tolerant systems to make sure that every component was redundant so that no single component failure could bring down the whole system. Chapter 6 shows how a software layer avoids single points of failure inside WSCs.

Fallacy *Hardware enhancements that increase performance also improve energy efficiency, or are at worst energy neutral.*

Esmaeilzadeh et al. (2011) measured SPEC2006 on just one core of a 2.67 GHz Intel Core i7 using Turbo mode (Section 1.5). Performance increased by a factor of 1.07 when the clock rate increased to 2.94 GHz (or a factor of 1.10), but the i7 used a factor of 1.37 more joules and a factor of 1.47 more watt hours!

Fallacy *Benchmarks remain valid indefinitely.*

Several factors influence the usefulness of a benchmark as a predictor of real performance, and some change over time. A big factor influencing the usefulness of a benchmark is its ability to resist "benchmark engineering" or "benchmarketing." Once a benchmark becomes standardized and popular, there is tremendous pressure to improve performance by targeted optimizations or by aggressive interpretation of the rules for running the benchmark. Short kernels or programs that spend their time in a small amount of code are particularly vulnerable.

For example, despite the best intentions, the initial SPEC89 benchmark suite included a small kernel, called matrix300, which consisted of eight different 300×300 matrix multiplications. In this kernel, 99% of the execution time was in a single line (see SPEC, 1989). When an IBM compiler optimized this inner loop

(using a good idea called *blocking*, discussed in Chapters 2 and 4), performance improved by a factor of 9 over a prior version of the compiler! This benchmark tested compiler tuning and was not, of course, a good indication of overall performance, nor of the typical value of this particular optimization.

Figure 1.19 shows that if we ignore history, we may be forced to repeat it. SPEC Cint2006 had not been updated for a decade, giving compiler writers substantial time to hone their optimizers to this suite. Note that the SPEC ratios of all benchmarks but libquantum fall within the range of 16–52 for the AMD computer and from 22 to 78 for Intel. Libquantum runs about 250 times faster on AMD and 7300 times faster on Intel! This "miracle" is a result of optimizations by the Intel compiler that automatically parallelizes the code across 22 cores and optimizes memory by using bit packing, which packs together multiple narrow-range integers to save memory space and thus memory bandwidth. If we drop this benchmark and recalculate the geometric means, AMD SPEC Cint2006 falls from 31.9 to 26.5 and Intel from 63.7 to 41.4. The Intel computer is now about 1.5 times as fast as the AMD computer instead of 2.0 if we include libquantum, which is surely closer to their real relative performances. SPECCPU2017 dropped libquantum.

To illustrate the short lives of benchmarks, Figure 1.17 on page 43 lists the status of all 82 benchmarks from the various SPEC releases; Gcc is the lone survivor from SPEC89. Amazingly, about 70% of all programs from SPEC2000 or earlier were dropped from the next release.

Fallacy *The rated mean time to failure of disks is 1,200,000 hours or almost 140 years, so disks practically never fail.*

The current marketing practices of disk manufacturers can mislead users. How is such an MTTF calculated? Early in the process, manufacturers will put thousands of disks in a room, run them for a few months, and count the number that fail. They compute MTTF as the total number of hours that the disks worked cumulatively divided by the number that failed.

One problem is that this number far exceeds the lifetime of a disk, which is commonly assumed to be five years or 43,800 hours. For this large MTTF to make some sense, disk manufacturers argue that the model corresponds to a user who buys a disk and then keeps replacing the disk every 5 years—the planned lifetime of the disk. The claim is that if many customers (and their great-grandchildren) did this for the next century, on average they would replace a disk 27 times before a failure, or about 140 years.

A more useful measure is the percentage of disks that fail, which is called the *annual failure rate*. Assume 1000 disks with a 1,000,000-hour MTTF and that the disks are used 24 hours a day. If you replaced failed disks with a new one having the same reliability characteristics, the number that would fail in a year (8760 hours) is

$$\text{Failed disks} = \frac{\text{Number of disks} \times \text{Time period}}{\text{MTTF}} = \frac{1000 \text{ disks} \times 8760 \text{ hours/drive}}{1{,}000{,}000 \text{ hours/failure}} = 9$$

Stated alternatively, 0.9% would fail per year, or 4.4% over a 5-year lifetime.

Moreover, those high numbers are quoted assuming limited ranges of temperature and vibration; if they are exceeded, then all bets are off. A survey of disk drives in real environments (Gray and van Ingen, 2005) found that 3%–7% of drives failed per year, for an MTTF of about 125,000–300,000 hours. An even larger study found annual disk failure rates of 2%–10% (Pinheiro et al., 2007). Therefore the real-world MTTF is about 2–10 times worse than the manufacturer's MTTF.

Fallacy *Peak performance tracks observed performance.*

The only universally true definition of peak performance is "the performance level a computer is guaranteed not to exceed." Figure 1.24 shows the percentage of peak performance for four programs on four multiprocessors. It varies from 5% to 58%. Since the gap is so large and can vary significantly by benchmark, peak performance is not generally useful in predicting observed performance.

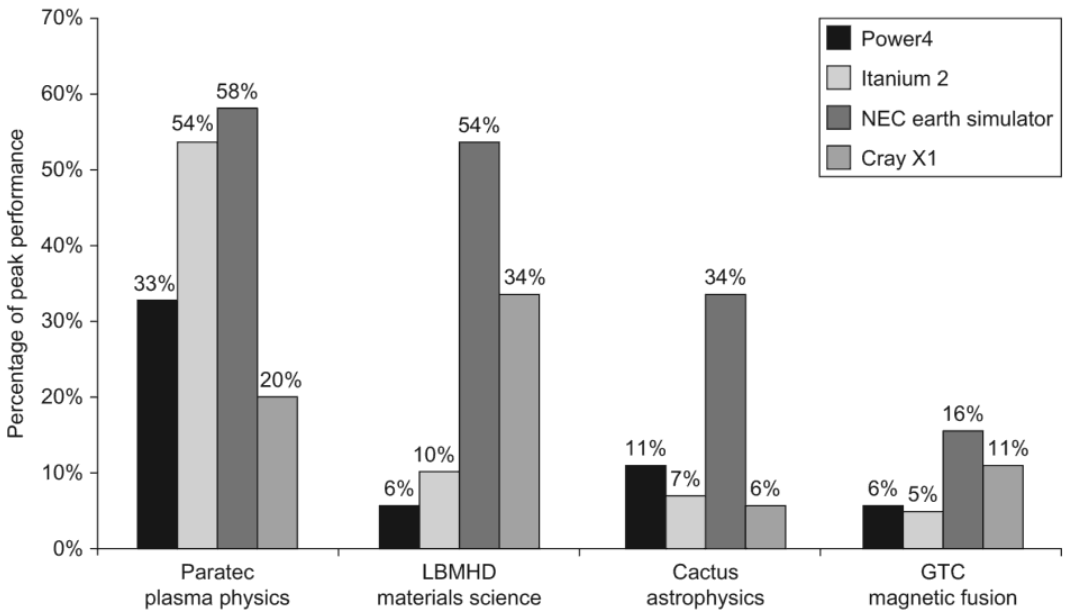


Figure 1.24 Percentage of peak performance for four programs on four multiprocessors scaled to 64 processors. The Earth Simulator and X1 are vector processors (see Chapter 4 and Appendix G). Not only did they deliver a higher fraction of peak performance, but they also had the highest peak performance and the lowest clock rates. Except for the Paratec program, the Power 4 and Itanium 2 systems delivered between 5% and 10% of their peak. From Oliker, L., Canning, A., Carter, J., Shalf, J., Ethier, S., 2004. Scientific computations on modern parallel vector systems. In: Proc. ACM/IEEE Conf. on Supercomputing, November 6–12, 2004, Pittsburgh, Penn., p. 10.

Pitfall *Fault detection can lower availability.*

This apparently ironic pitfall is because computer hardware has a fair amount of state that may not always be critical to proper operation. For example, it is not fatal if an error occurs in a branch predictor, because only performance may suffer.

In processors that try to exploit ILP aggressively, not all the operations are needed for correct execution of the program. Mukherjee et al. (2003) found that less than 30% of the operations were potentially on the critical path for the SPEC2000 benchmarks.

The same observation is true about programs. If a register is "dead" in a program—that is, the program will write the register before it is read again—then errors do not matter. If you were to crash the program upon detection of a transient fault in a dead register, it would lower availability unnecessarily.

The Sun Microsystems Division of Oracle lived this pitfall in 2000 with an L2 cache that included parity, but not error correction, in its Sun E3000 to Sun E10000 systems. The SRAMs they used to build the caches had intermittent faults, which parity detected. If the data in the cache were not modified, the processor would simply reread the data from the cache. Because the designers did not protect the cache with ECC (error-correcting code), the operating system had no choice but to report an error to dirty data and crash the program. Field engineers found no problems on inspection in more than 90% of the cases.

To reduce the frequency of such errors, Sun modified the Solaris operating system to "scrub" the cache by having a process that proactively wrote dirty data to memory. Because the processor chips did not have enough pins to add ECC, the only hardware option for dirty data was to duplicate the external cache, using the copy without the parity error to correct the error.

The pitfall is in detecting faults without providing a mechanism to correct them. These engineers are unlikely to design another computer without ECC on external caches.

1.12 Concluding Remarks

This chapter has introduced a number of concepts and provided a quantitative framework that we will expand on throughout the book. Starting with the last edition, energy efficiency is the constant companion to performance.

In Chapter 2, we start with the all-important area of memory system design. We will examine a wide range of techniques that conspire to make memory look infinitely large while still being as fast as possible. (Appendix B provides introductory material on caches for readers without much experience and background with them.) As in later chapters, we will see that hardware-software cooperation has become a key to high-performance memory systems, just as it has to high-performance pipelines. This chapter also covers virtual machines, an increasingly important technique for protection.

In Chapter 3, we look at ILP, of which pipelining is the simplest and most common form. Exploiting ILP is one of the most important techniques for building

high-speed uniprocessors. Chapter 3 begins with an extensive discussion of basic concepts that will prepare you for the wide range of ideas examined in both chapters. Chapter 3 uses examples that span about 40 years, drawing from one of the first supercomputers (IBM 360/91) to the fastest processors on the market in 2017. It emphasizes what is called the *dynamic* or *runtime approach* to exploiting ILP. It also talks about the limits to ILP ideas and introduces multithreading, which is further developed in both Chapters 4 and 5. Appendix C provides introductory material on pipelining for readers without much experience and background in pipelining. (We expect it to be a review for many readers, including those of our introductory text, *Computer Organization and Design: The Hardware/Software Interface*.)

Chapter 4 explains three ways to exploit data-level parallelism. The classic and oldest approach is vector architecture, and we start there to lay down the principles of SIMD design. (Appendix G goes into greater depth on vector architectures.) We next explain the SIMD instruction set extensions found in most desktop microprocessors today. The third piece is an in-depth explanation of how modern graphics processing units (GPUs) work. Most GPU descriptions are written from the programmer's perspective, which usually hides how the computer really works. This section explains GPUs from an insider's perspective, including a mapping between GPU jargon and more traditional architecture terms.

Chapter 5 focuses on the issue of achieving higher performance using multiple processors, or multiprocessors. Instead of using parallelism to overlap individual instructions, multiprocessing uses parallelism to allow multiple instruction streams to be executed simultaneously on different processors. Our focus is on the dominant form of multiprocessors, shared-memory multiprocessors, though we introduce other types as well and discuss the broad issues that arise in any multiprocessor. Here again we explore a variety of techniques, focusing on the important ideas first introduced in the 1980s and 1990s.

Chapter 6 introduces clusters and then goes into depth on WSCs, which computer architects help design. The designers of WSCs are the professional descendants of the pioneers of supercomputers, such as Seymour Cray, in that they are designing extreme computers. WSCs contain tens of thousands of servers, and the equipment and the building that holds them cost nearly $200 million. The concerns of price-performance and energy efficiency of the earlier chapters apply to WSCs, as does the quantitative approach to making decisions.

Chapter 7 is new to this edition. It introduces domain-specific architectures as the only path forward for improved performance and energy efficiency given the end of Moore's Law and Dennard scaling. It offers guidelines on how to build effective domain-specific architectures, introduces the exciting domain of deep neural networks, describes four recent examples that take very different approaches to accelerating neural networks, and then compares their cost-performance.

This book comes with an abundance of material online (see Preface for more details), both to reduce cost and to introduce readers to a variety of advanced topics. Figure 1.25 shows them all. Appendices A–C, which appear in the book, will be a review for many readers.

Appendix	Title
A	Instruction Set Principles
B	Review of Memory Hierarchies
C	Pipelining: Basic and Intermediate Concepts
D	Storage Systems
E	Embedded Systems
F	Interconnection Networks
G	Vector Processors in More Depth
H	Hardware and Software for VLIW and EPIC
I	Large-Scale Multiprocessors and Scientific Applications
J	Computer Arithmetic
K	Survey of Instruction Set Architectures
L	Advanced Concepts on Address Translation
M	Historical Perspectives and References

Figure 1.25 List of appendices.

In Appendix D, we move away from a processor-centric view and discuss issues in storage systems. We apply a similar quantitative approach, but one based on observations of system behavior and using an end-to-end approach to performance analysis. This appendix addresses the important issue of how to store and retrieve data efficiently using primarily lower-cost magnetic storage technologies. Our focus is on examining the performance of disk storage systems for typical I/O-intensive workloads, such as the OLTP benchmarks mentioned in this chapter. We extensively explore advanced topics in RAID-based systems, which use redundant disks to achieve both high performance and high availability. Finally, Appendix D introduces queuing theory, which gives a basis for trading off utilization and latency.

Appendix E applies an embedded computing perspective to the ideas of each of the chapters and early appendices.

Appendix F explores the topic of system interconnect broadly, including wide area and system area networks that allow computers to communicate.

Appendix H reviews VLIW hardware and software, which, in contrast, are less popular than when EPIC appeared on the scene just before the last edition.

Appendix I describes large-scale multiprocessors for use in high-performance computing.

Appendix J is the only appendix that remains from the first edition, and it covers computer arithmetic.

Appendix K provides a survey of instruction architectures, including the 80x86, the IBM 360, the VAX, and many RISC architectures, including ARM, MIPS, Power, RISC-V, and SPARC.

Appendix L is new and discusses advanced techniques for memory management, focusing on support for virtual machines and design of address translation

for very large address spaces. With the growth in cloud processors, these architectural enhancements are becoming more important.

We describe Appendix M next.

1.13 Historical Perspectives and References

Appendix M (available online) includes historical perspectives on the key ideas presented in each of the chapters in this text. These historical perspective sections allow us to trace the development of an idea through a series of machines or to describe significant projects. If you're interested in examining the initial development of an idea or processor or want further reading, references are provided at the end of each history. For this chapter, see Section M.2, "The Early Development of Computers," for a discussion on the early development of digital computers and performance measurement methodologies.

As you read the historical material, you'll soon come to realize that one of the important benefits of the youth of computing, compared to many other engineering fields, is that some of the pioneers are still alive—we can learn the history by simply asking them!

Case Studies and Exercises by Diana Franklin

Case Study 1: Chip Fabrication Cost

Concepts illustrated by this case study

- Fabrication Cost
- Fabrication Yield
- Defect Tolerance Through Redundancy

Many factors are involved in the price of a computer chip. Intel is spending $7 billion to complete its Fab 42 fabrication facility for 7 nm technology. In this case study, we explore a hypothetical company in the same situation and how different design decisions involving fabrication technology, area, and redundancy affect the cost of chips.

1.1 [10/10] <1.6> Figure 1.26 gives hypothetical relevant chip statistics that influence the cost of several current chips. In the next few exercises, you will be exploring the effect of different possible design decisions for the Intel chips.

Chip	Die Size (mm^2)	Estimated defect rate (per cm^2)	*N*	Manufacturing size (nm)	Transistors (billion)	Cores
BlueDragon	180	0.03	12	10	7.5	4
RedDragon	120	0.04	14	7	7.5	4
Phoenix[8]	200	0.04	14	7	12	8

Figure 1.26 Manufacturing cost factors for several hypothetical current and future processors.

a. [10] <1.6> What is the yield for the Phoenix chip?

b. [10] <1.6> Why does Phoenix have a higher defect rate than BlueDragon?

1.2 [20/20/20/20] <1.6> They will sell a range of chips from that factory, and they need to decide how much capacity to dedicate to each chip. Imagine that they will sell two chips. Phoenix is a completely new architecture designed with 7 nm technology in mind, whereas RedDragon is the same architecture as their 10 nm BlueDragon. Imagine that RedDragon will make a profit of $15 per defect-free chip. Phoenix will make a profit of $30 per defect-free chip. Each wafer has a 450 mm diameter.

a. [20] <1.6> How much profit do you make on each wafer of Phoenix chips?

b. [20] <1.6> How much profit do you make on each wafer of RedDragon chips?

c. [20] <1.6> If your demand is 50,000 RedDragon chips per month and 25,000 Phoenix chips per month, and your facility can fabricate 70 wafers a month, how many wafers should you make of each chip?

1.3 [20/20] <1.6> Your colleague at AMD suggests that, since the yield is so poor, you might make chips more cheaply if you released multiple versions of the same chip, just with different numbers of cores. For example, you could sell Phoenix^8, Phoenix^4, Phoenix^2, and Phoenix^1, which contain 8, 4, 2, and 1 cores on each chip, respectively. If all eight cores are defect-free, then it is sold as Phoenix^8. Chips with four to seven defect-free cores are sold as Phoenix^4, and those with two or three defect-free cores are sold as Phoenix^2. For simplification, calculate the yield for a single core as the yield for a chip that is 1/8 the area of the original Phoenix chip. Then view that yield as an independent probability of a single core being defect free. Calculate the yield for each configuration as the probability of at the corresponding number of cores being defect free.

a. [20] <1.6> What is the yield for a single core being defect free as well as the yield for Phoenix^4, Phoenix^2 and Phoenix^1?

b. [5] <1.6> Using your results from part a, determine which chips you think it would be worthwhile to package and sell, and why.

c. [10] <1.6> If it previously cost $20 dollars per chip to produce Phoenix^8, what will be the cost of the new Phoenix chips, assuming that there are no additional costs associated with rescuing them from the trash?

d. [20] <1.6> You currently make a profit of $30 for each defect-free Phoenix^8, and you will sell each Phoenix^4 chip for $25. How much is your profit per Phoenix^8 chip if you consider (i) the purchase price of Phoenix^4 chips to be entirely profit and (ii) apply the profit of Phoenix^4 chips to each Phoenix^8 chip in proportion to how many are produced? Use the yields calculated from part Problem 1.3a, not from problem 1.1a.

Case Study 2: Power Consumption in Computer Systems

Concepts illustrated by this case study

- Amdahl's Law
- Redundancy
- MTTF
- Power Consumption

Power consumption in modern systems is dependent on a variety of factors, including the chip clock frequency, efficiency, and voltage. The following exercises explore the impact on power and energy that different design decisions and use scenarios have.

1.4 [10/10/10/10] <1.5> A cell phone performs very different tasks, including streaming music, streaming video, and reading email. These tasks perform very different computing tasks. Battery life and overheating are two common problems for cell phones, so reducing power and energy consumption are critical. In this problem, we consider what to do when the user is not using the phone to its full computing capacity. For these problems, we will evaluate an unrealistic scenario in which the cell phone has no specialized processing units. Instead, it has a quad-core, general-purpose processing unit. Each core uses 0.5 W at full use. For email-related tasks, the quad-core is 8× as fast as necessary.

a. [10] <1.5> How much dynamic energy and power are required compared to running at full power? First, suppose that the quad-core operates for 1/8 of the time and is idle for the rest of the time. That is, the clock is disabled for 7/8 of the time, with no leakage occurring during that time. Compare total dynamic energy as well as dynamic power while the core is running.

b. [10] <1.5> How much dynamic energy and power are required using frequency and voltage scaling? Assume frequency and voltage are both reduced to 1/8 the entire time.

c. [10] <1.6, 1.9> Now assume the voltage may not decrease below 50% of the original voltage. This voltage is referred to as the *voltage floor*, and any voltage lower than that will lose the state. Therefore, while the frequency can keep decreasing, the voltage cannot. What are the dynamic energy and power savings in this case?

d. [10] <1.5> How much energy is used with a dark silicon approach? This involves creating specialized ASIC hardware for each major task and power

gating those elements when not in use. Only one general-purpose core would be provided, and the rest of the chip would be filled with specialized units. For email, the one core would operate for 25% the time and be turned completely off with power gating for the other 75% of the time. During the other 75% of the time, a specialized ASIC unit that requires 20% of the energy of a core would be running.

1.5 [10/10/10] <1.5> As mentioned in Exercise 1.4, cell phones run a wide variety of applications. We'll make the same assumptions for this exercise as the previous one, that it is 0.5 W per core and that a quad core runs email 3× as fast.

a. [10] <1.5> Imagine that 80% of the code is parallelizable. By how much would the frequency and voltage on a single core need to be increased in order to execute at the same speed as the four-way parallelized code?

b. [10] <1.5> What is the reduction in dynamic energy from using frequency and voltage scaling in part a?

c. [10] <1.5> How much energy is used with a dark silicon approach? In this approach, all hardware units are power gated, allowing them to turn off entirely (causing no leakage). Specialized ASICs are provided that perform the same computation for 20% of the power as the general-purpose processor. Imagine that each core is power gated. The video game requires two ASICS and two cores. How much dynamic energy does it require compared to the baseline of parallelized on four cores?

1.6 [10/10/10/10/10/20] <1.5,1.9> General-purpose processes are optimized for general-purpose computing. That is, they are optimized for behavior that is generally found across a large number of applications. However, once the domain is restricted somewhat, the behavior that is found across a large number of the target applications may be different from general-purpose applications. One such application is deep learning or neural networks. Deep learning can be applied to many different applications, but the fundamental building block of inference—using the learned information to make decisions—is the same across them all. Inference operations are largely parallel, so they are currently performed on graphics processing units, which are specialized more toward this type of computation, and not to inference in particular. In a quest for more performance per watt, Google has created a custom chip using tensor processing units to accelerate inference operations in deep learning.[1] This approach can be used for speech recognition and image recognition, for example. This problem explores the trade-offs between this process, a general-purpose processor (Haswell E5-2699 v3) and a GPU (NVIDIA K80), in terms of performance and cooling. If heat is not removed from the computer efficiently, the fans will blow hot air back onto the computer, not cold air. Note: The differences are more than processor—on-chip memory and DRAM also come into play. Therefore statistics are at a system level, not a chip level.

[1]Cite paper at this website: https://drive.google.com/file/d/0Bx4hafXDDq2EMzRNcy1vSUxtcEk/view.

a. [10] <1.9> If Google's data center spends 70% of its time on workload A and 30% of its time on workload B when running GPUs, what is the speedup of the TPU system over the GPU system?

b. [10] <1.9> If Google's data center spends 70% of its time on workload A and 30% of its time on workload B when running GPUs, what percentage of Max IPS does it achieve for each of the three systems?

c. [15] <1.5, 1.9> Building on (b), assuming that the power scales linearly from idle to busy power as IPS grows from 0% to 100%, what is the performance per watt of the TPU system over the GPU system?

d. [10] <1.9> If another data center spends 40% of its time on workload A, 10% of its time on workload B, and 50% of its time on workload C, what are the speedups of the GPU and TPU systems over the general-purpose system?

e. [10] <1.5> A cooling door for a rack costs $4000 and dissipates 14 kW (into the room; additional cost is required to get it out of the room). How many Haswell-, NVIDIA-, or Tensor-based servers can you cool with one cooling door, assuming TDP in Figures 1.27 and 1.28?

f. [20] <1.5> Typical server farms can dissipate a maximum of 200 W per square foot. Given that a server rack requires 11 square feet (including front and back clearance), how many servers from part (e) can be placed on a single rack, and how many cooling doors are required?

System	Chip	TDP	Idle power	Busy power
General-purpose	Haswell E5-2699 v3	504 W	159 W	455 W
Graphics processor	NVIDIA K80	1838 W	357 W	991 W
Custom ASIC	TPU	861 W	290 W	384 W

Figure 1.27 Hardware characteristics for general-purpose processor, graphical processing unit-based or custom ASIC-based system, including measured power (cite ISCA paper).

System	Chip	Throughput			% Max IPS		
		A	B	C	A	B	C
General-purpose	Haswell E5-2699 v3	5482	13,194	12,000	42%	100%	90%
Graphics processor	NVIDIA K80	13,461	36,465	15,000	37%	100%	40%
Custom ASIC	TPU	225,000	280,000	2000	80%	100%	1%

Figure 1.28 Performance characteristics for general-purpose processor, graphical processing unit-based or custom ASIC-based system on two neural-net workloads (cite ISCA paper). Workloads A and B are from published results. Workload C is a fictional, more general-purpose application.

Exercises

1.7 [10/15/15/10/10] <1.4, 1.5> One challenge for architects is that the design created today will require several years of implementation, verification, and testing before appearing on the market. This means that the architect must project what the technology will be like several years in advance. Sometimes, this is difficult to do.

a. [10] <1.4> According to the trend in device scaling historically observed by Moore's Law, the number of transistors on a chip in 2025 should be how many times the number in 2015?

b. [15] <1.5> The increase in performance once mirrored this trend. Had performance continued to climb at the same rate as in the 1990s, approximately what performance would chips have over the VAX-11/780 in 2025?

c. [15] <1.5> At the current rate of increase of the mid-2000s, what is a more updated projection of performance in 2025?

d. [10] <1.4> What has limited the rate of growth of the clock rate, and what are architects doing with the extra transistors now to increase performance?

e. [10] <1.4> The rate of growth for DRAM capacity has also slowed down. For 20 years, DRAM capacity improved by 60% each year. If 8 Gbit DRAM was first available in 2015, and 16 Gbit is not available until 2019, what is the current DRAM growth rate?

1.8 [10/10] <1.5> You are designing a system for a real-time application in which specific deadlines must be met. Finishing the computation faster gains nothing. You find that your system can execute the necessary code, in the worst case, twice as fast as necessary.

a. [10] <1.5> How much energy do you save if you execute at the current speed and turn off the system when the computation is complete?

b. [10] <1.5> How much energy do you save if you set the voltage and frequency to be half as much?

1.9 [10/10/20/20] <1.5> Server farms such as Google and Yahoo! provide enough compute capacity for the highest request rate of the day. Imagine that most of the time these servers operate at only 60% capacity. Assume further that the power does not scale linearly with the load; that is, when the servers are operating at 60% capacity, they consume 90% of maximum power. The servers could be turned off, but they would take too long to restart in response to more load. A new system has been proposed that allows for a quick restart but requires 20% of the maximum power while in this "barely alive" state.

a. [10] <1.5> How much power savings would be achieved by turning off 60% of the servers?

b. [10] <1.5> How much power savings would be achieved by placing 60% of the servers in the "barely alive" state?

c. [20] <1.5> How much power savings would be achieved by reducing the voltage by 20% and frequency by 40%?

d. [20] <1.5> How much power savings would be achieved by placing 30% of the servers in the "barely alive" state and 30% off?

1.10 [10/10/20] <1.7> Availability is the most important consideration for designing servers, followed closely by scalability and throughput.

a. [10] <1.7> We have a single processor with a failure in time (FIT) of 100. What is the mean time to failure (MTTF) for this system?

b. [10] <1.7> If it takes one day to get the system running again, what is the availability of the system?

c. [20] <1.7> Imagine that the government, to cut costs, is going to build a supercomputer out of inexpensive computers rather than expensive, reliable computers. What is the MTTF for a system with 1000 processors? Assume that if one fails, they all fail.

1.11 [20/20/20] <1.1, 1.2, 1.7> In a server farm such as that used by Amazon or eBay, a single failure does not cause the entire system to crash. Instead, it will reduce the number of requests that can be satisfied at any one time.

a. [20] <1.7> If a company has 10,000 computers, each with an MTTF of 35 days, and it experiences catastrophic failure only if 1/3 of the computers fail, what is the MTTF for the system?

b. [20] <1.1, 1.7> If it costs an extra $1000, per computer, to double the MTTF, would this be a good business decision? Show your work.

c. [20] <1.2> Figure 1.3 shows, on average, the cost of downtimes, assuming that the cost is equal at all times of the year. For retailers, however, the Christmas season is the most profitable (and therefore the most costly time to lose sales). If a catalog sales center has twice as much traffic in the fourth quarter as every other quarter, what is the average cost of downtime per hour during the fourth quarter and the rest of the year?

1.12 [20/10/10/10/15] <1.9> In this exercise, assume that we are considering enhancing a quad-core machine by adding encryption hardware to it. When computing encryption operations, it is 20 times faster than the normal mode of execution. We will define percentage of encryption as the percentage of time in the original execution that is spent performing encryption operations. The specialized hardware increases power consumption by 2%.

a. [20] <1.9> Draw a graph that plots the speedup as a percentage of the computation spent performing encryption. Label the y-axis "Net speedup" and label the x-axis "Percent encryption."

b. [10] <1.9> With what percentage of encryption will adding encryption hardware result in a speedup of 2?

c. [10] <1.9> What percentage of time in the new execution will be spent on encryption operations if a speedup of 2 is achieved?

d. [15] <1.9> Suppose you have measured the percentage of encryption to be 50%. The hardware design group estimates it can speed up the encryption hardware even more with significant additional investment. You wonder whether adding a second unit in order to support parallel encryption operations would be more useful. Imagine that in the original program, 90% of the encryption operations could be performed in parallel. What is the speedup of providing two or four encryption units, assuming that the parallelization allowed is limited to the number of encryption units?

1.13 [15/10] <1.9> Assume that we make an enhancement to a computer that improves some mode of execution by a factor of 10. Enhanced mode is used 50% of the time, measured as a percentage of the execution time *when the enhanced mode is in use*. Recall that Amdahl's Law depends on the fraction of the original, unenhanced execution time that could make use of enhanced mode. Thus we cannot directly use this 50% measurement to compute speedup with Amdahl's Law.

a. [15] <1.9> What is the speedup we have obtained from fast mode?

b. [10] <1.9> What percentage of the original execution time has been converted to fast mode?

1.14 [20/20/15] <1.9> When making changes to optimize part of a processor, it is often the case that speeding up one type of instruction comes at the cost of slowing down something else. For example, if we put in a complicated fast floating-point unit, that takes space, and something might have to be moved farther away from the middle to accommodate it, adding an extra cycle in delay to reach that unit. The basic Amdahl's Law equation does not take into account this trade-off.

a. [20] <1.9> If the new fast floating-point unit speeds up floating-point operations by, on average, 2x, and floating-point operations take 20% of the original program's execution time, what is the overall speedup (ignoring the penalty to any other instructions)?

b. [20] <1.9> Now assume that speeding up the floating-point unit slowed down data cache accesses, resulting in a 1.5x slowdown (or 2/3 speedup). Data cache accesses consume 10% of the execution time. What is the overall speedup now?

c. [15] <1.9> After implementing the new floating-point operations, what percentage of execution time is spent on floating-point operations? What percentage is spent on data cache accesses?

1.15 [10/10/20/20] <1.10> Your company has just bought a new 22-core processor, and you have been tasked with optimizing your software for this processor. You will run four applications on this system, but the resource requirements are not equal. Assume the system and application characteristics listed in Table 1.1.

Table 1.1 Four applications

Application	A	B	C	D
% resources needed	41	27	18	14
% parallelizable	50	80	60	90

The percentage of resources of assuming they are all run in serial. Assume that when you parallelize a portion of the program by X, the speedup for that portion is X.

a. [10] <1.10> How much speedup would result from running application A on the entire 22-core processor, as compared to running it serially?

b. [10] <1.10> How much speedup would result from running application D on the entire 22-core processor, as compared to running it serially?

c. [20] <1.10> Given that application A requires 41% of the resources, if we statically assign it 41% of the cores, what is the overall speedup if A is run parallelized but everything else is run serially?

d. [20] <1.10> What is the overall speedup if all four applications are statically assigned some of the cores, relative to their percentage of resource needs, and all run parallelized?

e. [10] <1.10> Given acceleration through parallelization, what new percentage of the resources are the applications receiving, considering only active time on their statically-assigned cores?

1.16 [10/20/20/20/25] <1.10> When parallelizing an application, the ideal speedup is speeding up by the number of processors. This is limited by two things: percentage of the application that can be parallelized and the cost of communication. Amdahl's Law takes into account the former but not the latter.

a. [10] <1.10> What is the speedup with *N* processors if 80% of the application is parallelizable, ignoring the cost of communication?

b. [20] <1.10> What is the speedup with eight processors if, for every processor added, the communication overhead is 0.5% of the original execution time.

c. [20] <1.10> What is the speedup with eight processors if, for every time the number of processors is doubled, the communication overhead is increased by 0.5% of the original execution time?

d. [20] <1.10> What is the speedup with *N* processors if, for every time the number of processors is doubled, the communication overhead is increased by 0.5% of the original execution time?

e. [25] <1.10> Write the general equation that solves this question: What is the number of processors with the highest speedup in an application in which P% of the original execution time is parallelizable, and, for every time the number of processors is doubled, the communication is increased by 0.5% of the original execution time?

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).

Computer Architecture. https://doi.org/10.1016/B978-0-12-811905-1.00002-X

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 several different multilevel memory hierarchies, including typical sizes and speeds of access. As Flash and next generation memory technologies continue to close the gap with disks in cost per bit, such technologies are likely to increasingly replace magnetic disks for secondary storage. As Figure 2.1 shows, these technologies are already used in many personal computers and increasingly in servers, where the advantages in performance, power, and density are significant.

Because fast memory is more 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 a cost per byte that is almost as low as the cheapest level of memory and a 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 secondary storage (disk or Flash) in the case of virtual memory.

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), assuming a single DRAM and a single memory bank. The reality is more complex because the processor request rate is not uniform, and the memory system typically has multiple banks of DRAMs and channels. Although the gap in access time increased significantly for many years, the lack of significant performance improvement in single processors has led to a slowdown in the growth of the gap between processors and DRAM.

Because high-end processors have multiple cores, the bandwidth requirements are greater than for single cores. Although single-core bandwidth has grown more slowly in recent years, the gap between CPU memory demand and DRAM bandwidth continues to grow as the numbers of cores grow. A modern high-end desktop processor such as the Intel Core i7 6700 can generate two data memory references per core each clock cycle. With four cores and a 4.2 GHz clock rate, the i7 can generate a peak of 32.8 billion 64-bit data memory references per second, in addition to a peak instruction demand of about 12.8 billion 128-bit instruction

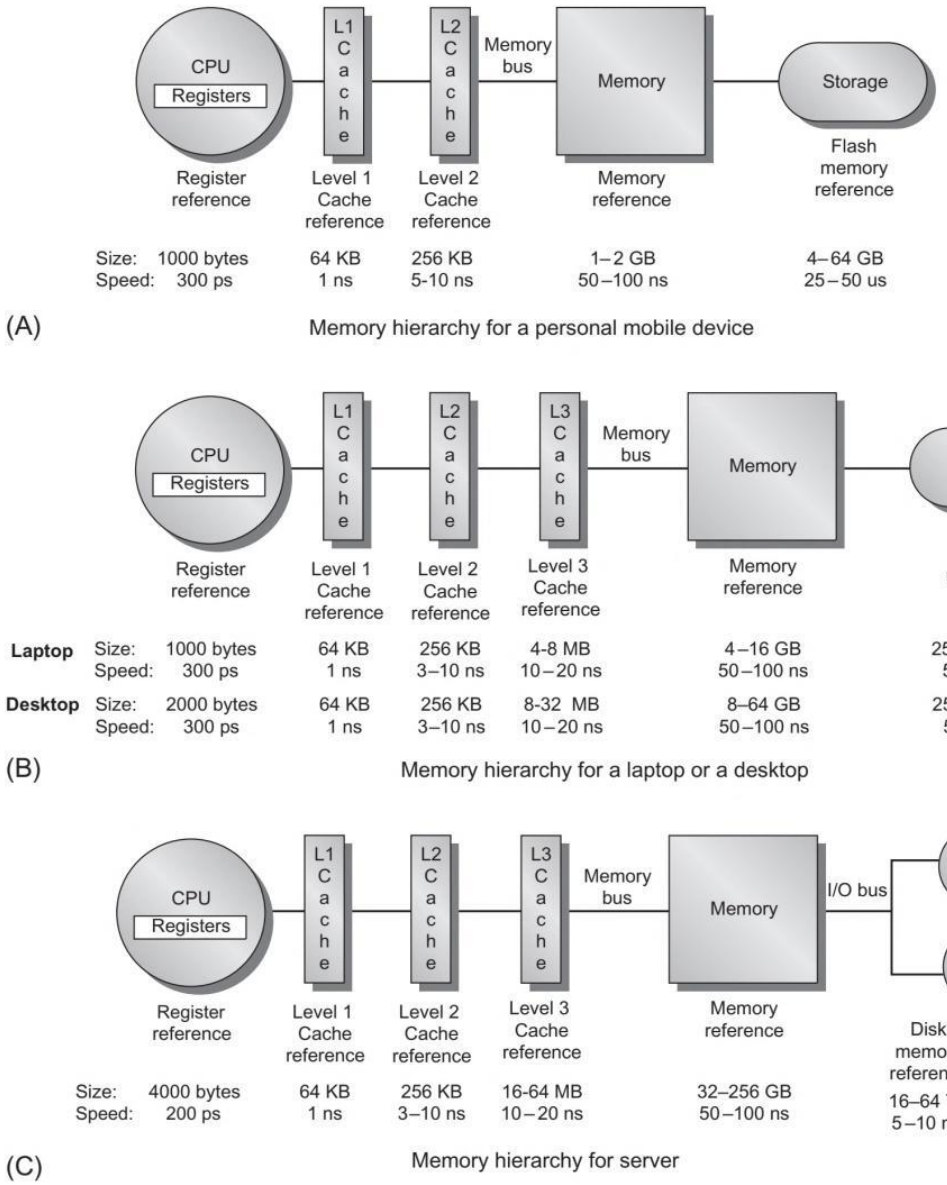


Figure 2.1 The levels in a typical memory hierarchy in a personal mobile device (PMD), such as a cell phone or tablet (A), in a laptop or desktop computer (B), and in a server (C). As we move farther away from the processor, the memory in the level below becomes slower and larger. Note that the time units change by a factor of 10^9 from picoseconds to milliseconds in the case of magnetic disks and that the size units change by a factor of 10^{10} from thousands of bytes to tens of terabytes. If we were to add warehouse-sized computers, as opposed to just servers, the capacity scale would increase by three to six orders of magnitude. Solid-state drives (SSDs) composed of Flash are used exclusively in PMDs, and heavily in both laptops and desktops. In many desktops, the primary storage system is SSD, and expansion disks are primarily hard disk drives (HDDs). Likewise, many servers mix SSDs and HDDs.

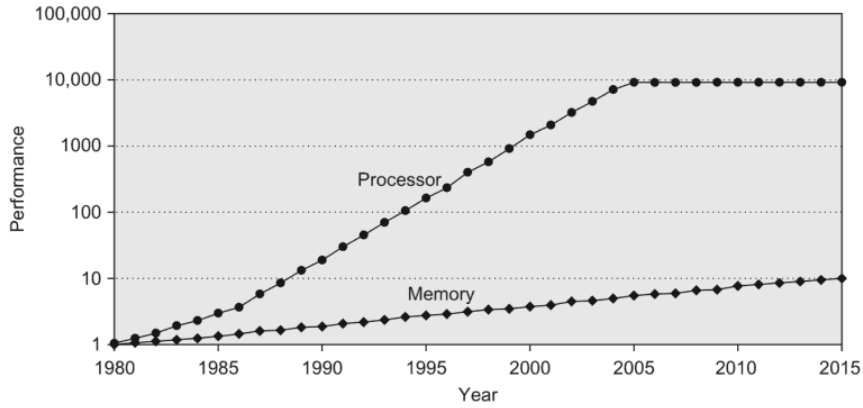


Figure 2.2 Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time. In mid-2017, AMD, Intel and Nvidia all announced chip sets using versions of HBM technology. 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 KiB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 2.4 on page 88). 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 only small improvements in processor performance (on a per-core basis) between 2005 and 2015. As you can see, until 2010 memory access times in DRAM improved slowly but consistently; since 2010 the improvement in access time has reduced, as compared with the earlier periods, although there have been continued improvements in bandwidth. See Figure 1.1 in Chapter 1 for more information.

references; this is a total peak demand bandwidth of 409.6 GiB/s! This incredible bandwidth is achieved by multiporting and pipelining the caches; by using three levels of caches, with two private levels per core and a shared L3; and by using a separate instruction and data cache at the first level. In contrast, the peak bandwidth for DRAM main memory, using two memory channels, is only 8% of the demand bandwidth (34.1 GiB/s). Upcoming versions are expected to have an L4 DRAM cache using embedded or stacked DRAM (see Sections 2.2 and 2.3).

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 60 MiB 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:

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

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 because 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 for 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 were to have 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.

Figure B.8 on page 24 shows the relative frequency of cache misses broken down by the three Cs. As mentioned in Appendix B, the three C's model is conceptual, and although its insights usually hold, it is not a definitive model for explaining the cache behavior of individual references.

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.

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

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

(This equation is often expressed in integers rather than fractions, as misses per 1000 instructions.)

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. 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 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 preceding six optimizations 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 preceding material 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 desktop or smaller server, the Intel Core i7 6700, as well as one designed for use in a PMD, the Arm Cortex-53, which is the basis for the processor used in several tablets and smartphones. Within each of these classes, there is a significant diversity in approach because of the intended use of the computer.

Although the i7 6700 has more cores and bigger caches than the Intel processors designed for mobile uses, the processors have similar architectures. A processor designed for small servers, such as the i7 6700, or larger servers, such as the Intel Xeon processors, typically is running a large number of concurrent processes, often for different users. Thus memory bandwidth becomes more important, and these processors offer larger caches and more aggressive memory systems to boost that bandwidth.

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 must consider both performance and energy consumption, which determines battery life. Before we dive into more advanced cache organizations and optimizations, one needs to understand the various memory technologies and how they are evolving.

2.2 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)

This section describes the technologies used in a memory hierarchy, specifically in building caches and main memory. These technologies are SRAM (static random-access memory), DRAM (dynamic random-access memory), and Flash. The last of these is used as an alternative to hard disks, but because its characteristics are based on semiconductor technology, it is appropriate to include in this section.

Using SRAM addresses the need to minimize access time to caches. When a cache miss occurs, however, we need to move the data from the main memory as quickly as possible, which requires a high bandwidth memory. This high memory bandwidth can be achieved by organizing the many DRAM chips that make up the main memory into multiple memory banks and by making the memory bus wider, or by doing both.

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, we need to introduce some terminology.

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. PMDs must balance power and performance, and because they have more modest storage needs, PMDs use Flash rather than disk drives, a decision increasingly being followed by desktop computers as well.

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—thus 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. In high-end server chips, there may be as many as 24 cores and up to 60 MiB of cache; such systems are often configured with 128–256 GiB of DRAM per processor chip. The access times for large, third-level, on-chip caches are typically two to eight times that of a second-level cache. Even so, the L3 access time is usually at least five times faster than a DRAM access.

On-chip, cache SRAMs are normally organized with a width that matches the block size of the cache, with the tags stored in parallel to each block. This allows an entire block to be read out or written into a single cycle. This capability is particularly useful when writing data fetched after a miss into the cache or when writing back a block that must be evicted from the cache. The access time to the cache (ignoring the hit detection and selection in a set associative cache) is proportional to the number of blocks in the cache, whereas the energy consumption depends both on the number of bits in the cache (static power) and on the number of blocks (dynamic power). Set associative caches reduce the initial access time to the memory because the size of the memory is smaller, but increase the time for hit detection and block selection, a topic we will cover in Section 2.3.

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

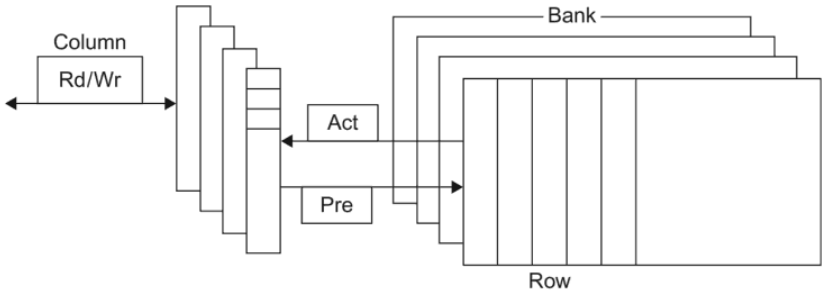


Figure 2.3 Internal organization of a DRAM. Modern DRAMs are organized in banks, up to 16 for DDR4. Each bank consists of a series of rows. Sending an ACT (Activate) command opens a bank and a row and loads the row into a row buffer. When the row is in the buffer, it can be transferred by successive column addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR4) or by specifying a block transfer and the starting address. The Precharge commend (PRE) closes the bank and row and readies it for a new access. Each command, as well as block transfers, are synchronized with a clock. See the next section discussing SDRAM. The row and column signals are sometimes called RAS and CAS, based on the original names of the signals.

cutting the number of address pins in half. Figure 2.3 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, because 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, which effectively acts as a capacitor, to store a bit. This has two implications: first, the sensing wires that detect the charge must be precharged, which sets them "halfway" between a logical 0 and 1, allowing the small charge stored in the cell to cause a 0 or 1 to be detected by the sense amplifiers. On reading, a row is placed into a row buffer, where CAS signals can select a portion of the row to read out from the DRAM. Because reading a row destroys the information, it must be written back when the row is no longer needed. This write back happens in overlapped fashion, but in early DRAMs, it meant that the cycle time before a new row could be read was larger than the time to read a row and access a portion of that row.

In addition, to prevent loss of information as the charge in a cell leaks away (assuming it is not read or written), each bit must be "refreshed" periodically. Fortunately, all the bits in a row can be refreshed simultaneously just by reading that row and writing it back. Therefore every DRAM in the memory system must access every row within a certain time window, such as 64 ms. DRAM 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 a row activation and a precharge that also writes the row back (which takes

roughly 2/3 of the time to get a datum because no column select is needed), and this is required for each row of the DRAM. Because the memory matrix in a DRAM is conceptually square, the number of steps 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. In fact, with SDRAMs, a DRAM controller (usually on the processor chip) tries to optimize accesses by avoiding opening new rows and using block transfer when possible. Refresh adds another unpredictable factor.

Amdahl suggested as a rule of thumb that memory capacity should grow linearly with processor speed to keep a balanced system. Thus a 1000 MIPS processor should have 1000 MiB 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. The slower performance improvements arise primarily because of smaller decreases in the row access time, which is determined by issues such as power limitations and the charge capacity (and thus the size) of an individual memory cell. Before we discuss these performance trends in more detail, we need to describe the major changes that occurred in DRAMs starting in the mid-1990s.

Improving Memory Performance Inside a DRAM Chip: SDRAMs

Although very early DRAMs included a buffer allowing multiple column accesses to a single row, without requiring a new row access, they used an asynchronous interface, which meant that every column access and transfer involved overhead to synchronize with the controller. In the mid-1990s, designers added a clock signal to the DRAM interface so that the repeated transfers would not bear that overhead, thereby creating *synchronous DRAM* (SDRAM). In addition to reducing overhead, SDRAMs allowed the addition of a burst transfer mode where multiple transfers can occur without specifying a new column address. Typically, eight or more 16-bit transfers can occur without sending any new addresses by placing the DRAM in burst mode. The inclusion of such burst mode transfers has meant that there is a significant gap between the bandwidth for a stream of random accesses versus access to a block of data.

To overcome the problem of getting more bandwidth from the memory as DRAM density increased, DRAMS were made wider. Initially, they offered a four-bit transfer mode; in 2017, DDR2, DDR3, and DDR DRAMS had up to 4, 8, or 16 bit buses.

In the early 2000s, a further innovation was introduced: *double data rate* (DDR), which allows a DRAM to transfer data both on the rising and the falling edge of the memory clock, thereby doubling the peak data rate.

Finally, SDRAMs introduced *banks* to help with power management, improve access time, and allow interleaved and overlapped accesses to different banks.

Access to different banks can be overlapped with each other, and each bank has its own row buffer. 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 a subsequent access specifies the same row for an open bank, the access can happen quickly, sending only the column address.

To initiate a new access, the DRAM controller sends a bank and row number (called *Activate* in SDRAMs and formerly called RAS—row select). That command opens the row and reads the entire row into a buffer. A column address can then be sent, and the SDRAM can transfer one or more data items, depending on whether it is a single item request or a burst request. Before accessing a new row, the bank must be precharged. If the row is in the same bank, then the precharge delay is seen; however, if the row is in another bank, closing the row and precharging can overlap with accessing the new row. In synchronous DRAMs, each of these command cycles requires an integral number of clock cycles.

From 1980 to 1995, DRAMs scaled with Moore's Law, doubling capacity every 18 months (or a factor of 4 in 3 years). From the mid-1990s to 2010, capacity increased more slowly with roughly 26 months between a doubling. From 2010 to 2016, capacity only doubled! Figure 2.4 shows the capacity and access time for various generations of DDR SDRAMs. From DDR1 to DDR3, access times improved by a factor of about 3, or about 7% per year. DDR4 improves power and bandwidth over DDR3, but has similar access latency.

As Figure 2.4 shows, DDR is a sequence of standards. DDR2 lowers power from DDR1 by dropping the voltage from 2.5 to 1.8 V and offers higher clock rates: 266, 333, and 400 MHz. DDR3 drops voltage to 1.5 V and has a maximum clock speed of 800 MHz. (As we discuss in the next section, GDDR5 is a graphics

			Best case access time (no precharge)			Precharge needed
Production year	Chip size	DRAM type	RAS time (ns)	CAS time (ns)	Total (ns)	Total (ns)
2000	256M bit	DDR1	21	21	42	63
2002	512M bit	DDR1	15	15	30	45
2004	1G bit	DDR2	15	15	30	45
2006	2G bit	DDR2	10	10	20	30
2010	4G bit	DDR3	13	13	26	39
2016	8G bit	DDR4	13	13	26	39

Figure 2.4 Capacity and access times for DDR SDRAMs by year of production. Access time is for a random memory word and assumes a new row must be opened. If the row is in a different bank, we assume the bank is precharged; if the row is not open, then a precharge is required, and the access time is longer. As the number of banks has increased, the ability to hide the precharge time has also increased. DDR4 SDRAMs were initially expected in 2014, but did not begin production until early 2016.

Standard	I/O clock rate	M transfers/s	DRAM name	MiB/s/DIMM	DIMM name
DDR1	133	266	DDR266	2128	PC2100
DDR1	150	300	DDR300	2400	PC2400
DDR1	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	1333	2666	DDR4-2666	21,300	PC21300

Figure 2.5 Clock rates, bandwidth, and names of DDR DRAMS and DIMMs in 2016. 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. DDR4 saw significant first use in 2016.

RAM and is based on DDR3 DRAMs.) DDR4, which shipped in volume in early 2016, but was expected in 2014, drops the voltage to 1–1.2 V and has a maximum expected clock rate of 1600 MHz. DDR5 is unlikely to reach production quantities until 2020 or later.

With the introduction of DDR, memory designers increasing focused on bandwidth, because improvements in access time were difficult. Wider DRAMs, burst transfers, and double data rate all contributed to rapid increases in memory bandwidth. DRAMs are commonly sold on small boards called *dual inline memory modules* (DIMMs) that contain 4–16 DRAM chips and that are normally organized to be 8 bytes wide (+ ECC) for desktop and server systems. When DDR SDRAMs are packaged as DIMMs, they are confusingly labeled by the peak *DIMM* bandwidth. Therefore the DIMM name PC3200 comes from 200 MHz $\times$ 2 $\times$ 8 bytes, or 3200 MiB/s; it is populated with DDR SDRAM chips. Sustaining the confusion, the chips themselves are labeled with *the number of bits per second* rather than their clock rate, so a 200 MHz DDR chip is called a DDR400. Figure 2.5 shows the relationships' I/O clock rate, transfers per second per chip, chip bandwidth, chip name, DIMM bandwidth, and DIMM name.

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 DDR4 SDRAMs, the operating voltage has dropped to 1.2 V, significantly reducing power versus DDR2 and DDR3 SDRAMs. The addition of banks also reduced power because only the row in a single bank is read.

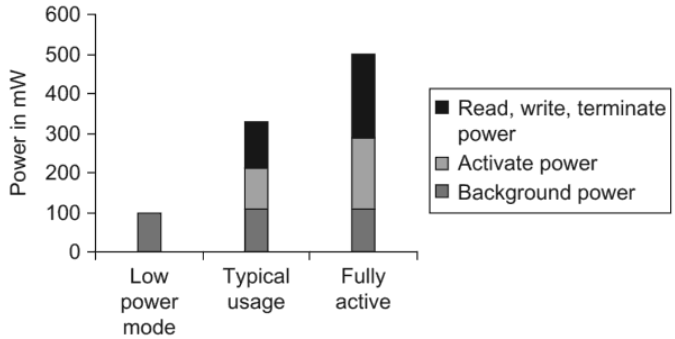


Figure 2.6 Power consumption for a DDR3 SDRAM operating under three conditions: low-power (shutdown) mode, typical system mode (DRAM is active 30% of the time for reads and 15% for writes), and fully active mode, where the DRAM is continuously reading or writing. Reads and writes assume bursts of eight transfers. These data are based on a Micron 1.5V 2GB DDR3-1066, although similar savings occur in DDR4 SDRAMs.

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 2.6 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 delay is 200 SDRAM clock cycles.

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. Because 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.

Packaging Innovation: Stacked or Embedded DRAMs

The newest innovation in 2017 in DRAMs is a packaging innovation, rather than a circuit innovation. It places multiple DRAMs in a stacked or adjacent fashion embedded within the same package as the processor. (Embedded DRAM also is used to refer to designs that place DRAM on the processor chip.) Placing the DRAM and processor in the same package lowers access latency (by shortening the delay between the DRAMs and the processor) and potentially increases bandwidth by allowing more and faster connections between the processor and DRAM; thus several producers have called it *high bandwidth memory (HBM)*.

One version of this technology places the DRAM die directly on the CPU die using solder bump technology to connect them. Assuming adequate heat management, multiple DRAM dies can be stacked in this fashion. Another approach stacks only DRAMs and abuts them with the CPU in a single package using a substrate (interposer) containing the connections. Figure 2.7 shows these two different interconnection schemes. Prototypes of HBM that allow stacking of up to eight chips have been demonstrated. With special versions of SDRAMs, such a package could contain 8 GiB of memory and have data transfer rates of 1 TB/s. The 2.5D technique is currently available. Because the chips must be specifically manufactured to stack, it is quite likely that most early uses will be in high-end server chipsets.

In some applications, it may be possible to internally package enough DRAM to satisfy the needs of the application. For example, a version of an Nvidia GPU used as a node in a special-purpose cluster design is being developed using HBM, and it is likely that HBM will become a successor to GDDR5 for higher-end applications. In some cases, it may be possible to use HBM as main memory, although the cost limitations and heat removal issues currently rule out this technology for some embedded applications. In the next section, we consider the possibility of using HBM as an additional level of cache.

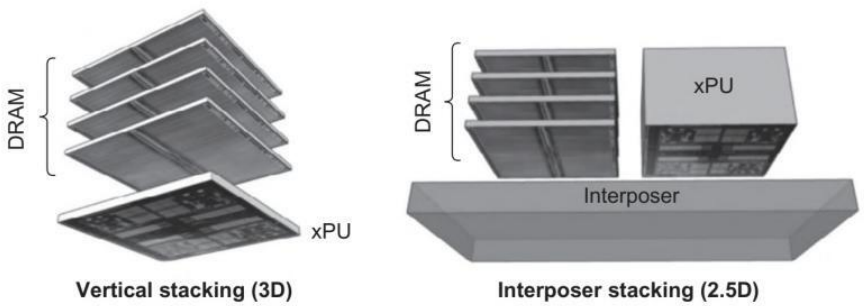


Figure 2.7 Two forms of die stacking. The 2.5D form is available now. 3D stacking is under development and faces heat management challenges due to the CPU.

Flash Memory

Flash memory is a type of EEPROM (electronically 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. We focus on NAND Flash, which has higher density than NOR Flash and is more suitable for large-scale nonvolatile memories; the drawback is that access is sequential and writing is slower, as we explain below.

Flash is used as the secondary 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 greater extent than it might have to do in a desktop or server with a main memory that might be 10–100 times larger.

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

1. Reads to Flash are sequential and read an entire page, which can be 512 bytes, 2 KiB, or 4 KiB. Thus NAND Flash has a long delay to access the first byte from a random address (about 25 μS), but can supply the remainder of a page block at about 40 MiB/s. By comparison, a DDR4 SDRAM takes about 40 ns to the first byte and can transfer the rest of the row at 4.8 GiB/s. Comparing the time to transfer 2 KiB, NAND Flash takes about 75 μS, while DDR SDRAM takes less than 500 ns, making Flash about 150 times slower. Compared to magnetic disk, however, a 2 KiB read from Flash is 300 to 500 times faster. From these numbers, we can see why Flash is not a candidate to replace DRAM for main memory, but is a candidate to replace magnetic disk.
2. Flash memory must be erased (thus the name flash for the "flash" erase process) before it is overwritten, and it is erased in blocks rather than individual bytes or words. This requirement means that 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. For writing, Flash is about 1500 times slower then SDRAM, and about 8–15 times as fast as magnetic disk.
3. Flash memory is nonvolatile (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).
4. Flash memory limits the number of times that any given block can be written, 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. This technique, called *write leveling*, is handled by Flash memory controllers.
5. High-density NAND Flash is cheaper than SDRAM but more expensive than disks: roughly $2/GiB for Flash, $20 to $40/GiB for SDRAM, and $0.09/GiB for magnetic disks. In the past five years, Flash has decreased in cost at a rate that is almost twice as fast as that of magnetic disks.

Like DRAM, Flash chips include redundant blocks to allow chips with small numbers of defects to be used; the remapping of blocks is handled in the Flash chip. Flash controllers handle page transfers, provide caching of pages, and handle write leveling.

The rapid improvements in high-density Flash have been critical to the development of low-power PMDs and laptops, but they have also significantly changed both desktops, which increasingly use solid state disks, and large servers, which often combine disk and Flash-based storage.

Phase-Change Memory Technology

Phase-change memory (PCM) has been an active research area for decades. The technology typically uses a small heating element to change the state of a bulk substrate between its crystalline form and an amorphous form, which have different resistive properties. Each bit corresponds to a crosspoint in a two-dimensional network that overlays the substrate. Reading is done by sensing the resistance between an x and y point (thus the alternative name *memristor*), and writing is accomplished by applying a current to change the phase of the material. The absence of an active device (such as a transistor) should lead to lower costs and greater density than that of NAND Flash.

In 2017 Micron and Intel began delivering Xpoint memory chips that are believed to be based on PCM. The technology is expected to have much better write durability than NAND Flash and, by eliminating the need to erase a page before writing, achieve an increase in write performance versus NAND of up to a factor of ten. Read latency is also better than Flash by perhaps a factor of 2–3. Initially, it is expected to be priced slightly higher than Flash, but the advantages in write performance and write durability may make it attractive, especially for SSDs. Should this technology scale well and be able to achieve additional cost reductions, it may be the solid state technology that will depose magnetic disks, which have reigned as the primary bulk nonvolatile store for more than 50 years.

Enhancing Dependability in Memory Systems

Large caches and main memories significantly increase the possibility of errors occurring both during the fabrication process and dynamically during operation. Errors that arise from a change in circuitry and are repeatable are called *hard errors* or *permanent faults.* Hard errors can occur during fabrication, as well as from a circuit change during operation (e.g., failure of a Flash memory cell after many writes). All DRAMs, Flash memory, and most 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. Dynamic errors, which are changes to a cell's contents, not a change in the circuitry, are called *soft errors* or *transient faults*.

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 GiB 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: 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 GiB) 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. Therefore Chipkill is a requirement for the 50,000–100,00 servers in warehouse-scale computers (see Section 6.8 of Chapter 6).

2.3 Ten Advanced Optimizations of Cache Performance

The preceding average memory access time formula 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 10 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 because of 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, and the final one depends on HBM. We will conclude with a summary of the implementation complexity and the performance benefits of the 10 techniques presented in Figure 2.18 on page 113. Because 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, because of 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 topic 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 a function of cache size, associativity, number of read/write ports, and more complex parameters. Figure 2.8 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 as fast as four-way and four-way is 1.4

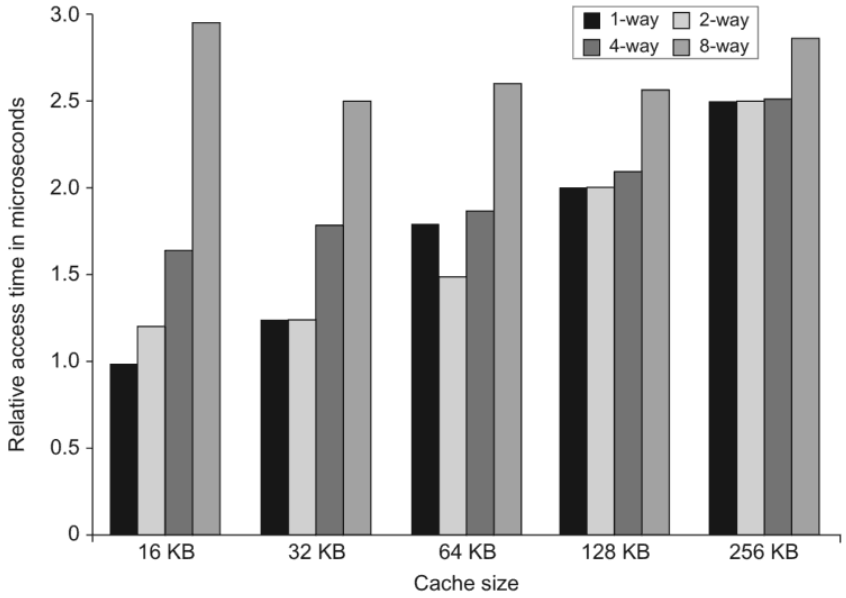


Figure 2.8 Relative access times generally increase as cache size and associativity are increased. These data come from the CACTI model 6.5 by Tarjan et al. (2005). The data assume typical embedded SRAM technology, a single bank, and 64-byte blocks. The assumptions about cache layout and the complex 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 KiB with two-way set associativity versus direct mapping. Similarly, the results with eight-way set associativity generate unusual behavior as cache size is increased. Because such observations are highly dependent on technology and detailed design assumptions, tools such as CACTI serve to reduce the search space. These results are relative; nonetheless, they are likely to shift as we move to more recent and denser semiconductor technologies.

times as fast as eight-way. Of course, these estimates depend on technology as well as the size of the cache, and CACTI must be carefully aligned with the technology; Figure 2.8 shows the relative tradeoffs for one technology.

Example Using the data in Figure B.8 in Appendix B and Figure 2.8, determine whether a 32 KiB four-way set associative L1 cache has a faster memory access time than a 32 KiB 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,

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

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:

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

Clearly, the higher associativity looks like a bad trade-off; however, because 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 2.9 shows. The energy cost of higher associativity ranges from more than a factor of 2 to negligible in caches of 128 or 256 KiB when going from direct mapped to two-way set associative.

As energy consumption has become critical, designers have focused on ways to reduce the energy needed for cache access. In addition to associativity, the other key factor in determining the energy used in a cache access is the number of blocks in the cache because it determines the number of "rows" that are accessed. A designer could reduce the number of rows by increasing the block size (holding total cache size constant), but this could increase the miss rate, especially in smaller L1 caches.

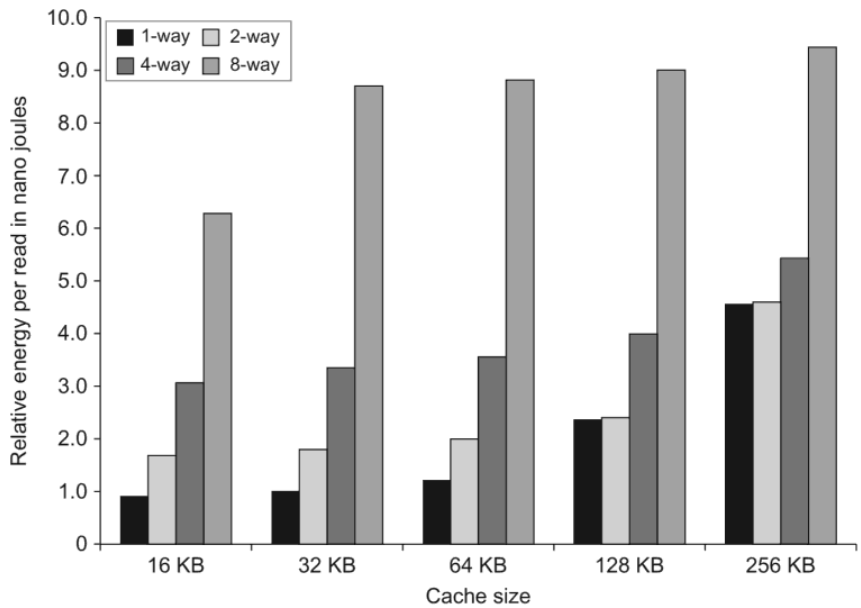


Figure 2.9 Energy consumption per read increases as cache size and associativity are increased. As in the previous figure, CACTI is used for the modeling with the same technology parameters. The large penalty for eight-way set associative caches is due to the cost of reading out eight tags and the corresponding data in parallel.

An alternative is to organize the cache in banks so that an access activates only a portion of the cache, namely the bank where the desired block resides. The primary use of multibanked caches is to increase the bandwidth of the cache, an optimization we consider shortly. Multibanking also reduces energy because less of the cache is accessed. The L3 caches in many multicores are logically unified, but physically distributed, and effectively act as a multibanked cache. Based on the address of a request, only one of the physical L3 caches (a bank) is actually accessed. We discuss this organization further in Chapter 5.

In recent designs, there are three other factors that have led to the use of higher associativity in first-level caches despite the energy and access time costs. First, many processors take at least 2 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.

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 in that clock cycle, only a single tag comparison is performed 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 was used in several ARM processors, which have 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, because 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 et al. (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; however, as energy concerns have mounted, schemes that do not require powering up the entire cache make increasing sense.

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 preceding study.

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 four-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 Access and Multibanked Caches to Increase Bandwidth

These optimizations increase cache bandwidth either by pipelining the cache access or by widening the cache with multiple banks to allow multiple accesses per clock; these optimizations are the dual to the superpipelined and superscalar approaches to increasing instruction throughput. These optimizations are primarily targeted at L1, where access bandwidth constrains instruction throughput. Multiple banks are also used in L2 and L3 caches, but primarily as a power-management technique.

Pipelining L1 allows a higher clock cycle, at the cost of increased latency. 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 clock cycles; and for the Pentium 4, which became available in 2000, and the current Intel Core i7, it takes 4 clock cycles. Pipelining the instruction cache effectively increases the number of pipeline stages,

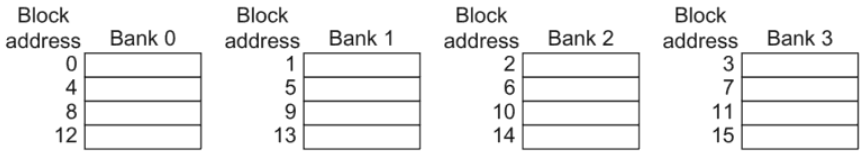


Figure 2.10 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per block, each of these addresses would be multiplied by 64 to get byte addressing.

leading to a greater penalty on mispredicted branches. Correspondingly, pipelining the data cache leads to more clock cycles between issuing the load and using the data (see Chapter 3). Today, all processors use some pipelining of L1, if only for the simple case of separating the access and hit detection, and many high-speed processors have three or more levels of cache pipelining.

It is easier to pipeline the instruction cache than the data cache because the processor can rely on high performance branch prediction to limit the latency effects. Many superscalar processors can issue and execute more than one memory reference per clock (allowing a load or store is common, and some processors allow multiple loads). To handle multiple data cache accesses per clock, we can divide the cache into independent banks, each supporting an independent access. Banks were originally used to improve performance of main memory and are now used inside modern DRAM chips as well as with caches. The Intel Core i7 has four banks in L1 (to support up to 2 memory accesses per clock).

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, which is 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 2.10 shows this interleaving. Multiple banks also are a way to reduce power consumption in both caches and DRAM.

Multiple banks are also useful in L2 or L3 caches, but for a different reason. With multiple banks in L2, we can handle more than one outstanding L1 miss, if the banks do not conflict. This is a key capability to support nonblocking caches, our next optimization. The L2 in the Intel Core i7 has eight banks, while Arm Cortex processors have used L2 caches with 1–4 banks. As mentioned earlier, multibanking can also reduce energy consumption.

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 *lockup-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 processors) usually support both, whereas many lower-end processors 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 KiB caches with a 14-cycle miss penalty (appropriate for the early 1990s). 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 et al. (2011) updated this study to use a multilevel cache, more modern assumptions about miss penalties, and the larger and more demanding SPECCPU2006 benchmarks. The study was done assuming a model based on a single core of an Intel i7 (see Section 2.6) running the SPECCPU2006 benchmarks. Figure 2.11 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 KiB data caches: 5.2% for floating-point programs with a direct-mapped cache, 4.9% for the 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

$$\text{Miss rate}_{\text{DM}} \times \text{Miss penalty} = 5.2\% \times 10 = 0.52$$
$$\text{Miss rate}_{\text{2-way}} \times \text{Miss penalty} = 4.9\% \times 10 = 0.49$$

The cache access latency (including stalls) for two-way associativity is 0.49/0.52 or 94% of direct-mapped cache. Figure 2.11 caption indicates that a hit under one miss reduces the average data cache access latency for floating-point programs to 87.5% of a blocking cache. Therefore, for floating-point programs, the

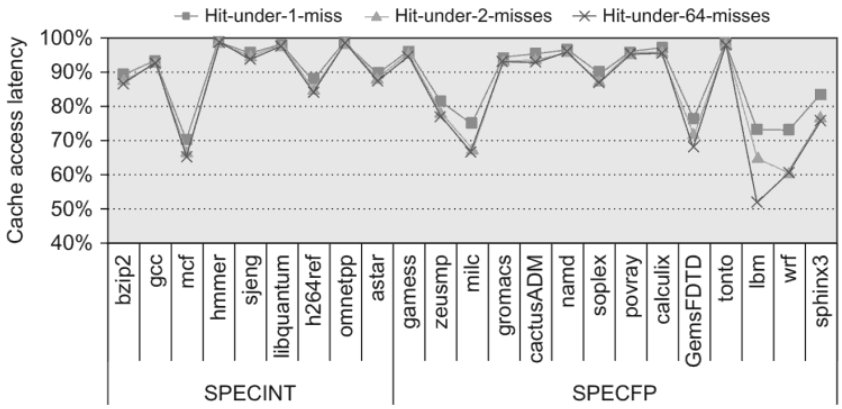


Figure 2.11 The effectiveness of a nonblocking cache is evaluated by allowing 1, 2, or 64 hits under a cache miss with 9 SPECINT (on the left) and 9 SPECFP (on the right) benchmarks. The data memory system modeled after the Intel i7 consists of a 32 KiB L1 cache with a four-cycle access latency. The L2 cache (shared with instructions) is 256 KiB with a 10-clock cycle access latency. The L3 is 2 MiB 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.

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

$$\text{Miss rate}_{\text{DM}} \times \text{Miss penalty} = 3.5\% \times 10 = 0.35$$
$$\text{Miss rate}_{\text{2-way}} \times \text{Miss penalty} = 3.2\% \times 10 = 0.32$$

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 thus to calculate the average memory access time. The effective miss penalty is not the sum of the misses but the nonoverlapped 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, because the miss must initiate at the highest level cache.
- The latency of the memory system.

The following simplified example illustrates 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 GiB/s. 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. Because 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, because we cannot start work on those colliding references; the memory system needs more independent references, not fewer! To approximate, we can simply assume that half the memory references do not have to 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 2.11.

Implementing a Nonblocking Cache

Although nonblocking caches have the potential to improve performance, they are nontrivial to implement. Two initial types of challenges arise: arbitrating contention between hits and misses, and tracking outstanding misses so that we know when loads or stores can proceed. Consider the first problem. In a blocking cache, misses cause the processor to stall and no further accesses to the cache will occur

until the miss is handled. In a nonblocking cache, however, hits can collide with misses returning from the next level of the memory hierarchy. If we allow multiple outstanding misses, which almost all recent processors do, it is even possible for misses to collide. These collisions must be resolved, usually by first giving priority to hits over misses, and second by ordering colliding misses (if they can occur).

The second problem arises because we need to track multiple outstanding misses. In a blocking cache, we always know which miss is returning, because only one can be outstanding. In a nonblocking cache, this is rarely true. At first glance, you might think that misses always return in order, so that a simple queue could be kept to match a returning miss with the longest outstanding request. Consider, however, a miss that occurs in L1. It may generate either a hit or miss in L2; if L2 is also nonblocking, then the order in which misses are returned to L1 will not necessarily be the same as the order in which they originally occurred. Multicore and other multiprocessor systems that have nonuniform cache access times also introduce this complication.

When a miss returns, the processor must know which load or store caused the miss, so that instruction can now go forward; and it must know where in the cache the data should be placed (as well as the setting of tags for that block). In recent processors, this information is kept in a set of registers, typically called the *Miss Status Handling Registers (MSHRs)*. If we allow n outstanding misses, there will be n MSHRs, each holding the information about where a miss goes in the cache and the value of any tag bits for that miss, as well as the information indicating which load or store caused the miss (in the next chapter, you will see how this is tracked). Thus, when a miss occurs, we allocate an MSHR for handling that miss, enter the appropriate information about the miss, and tag the memory request with the index of the MSHR. The memory system uses that tag when it returns the data, allowing the cache system to transfer the data and tag information to the appropriate cache block and "notify" the load or store that generated the miss that the data is now available and that it can resume operation. Nonblocking caches clearly require extra logic and thus have some cost in energy. It is difficult, however, to assess their energy costs exactly because they may reduce stall time, thereby decreasing execution time and resulting energy consumption.

In addition to the preceding issues, multiprocessor memory systems, whether within a single chip or on multiple chips, must also deal with complex implementation issues related to memory coherency and consistency. Also, because cache misses are no longer atomic (because the request and response are split and may be interleaved among multiple requests), there are possibilities for deadlock. For the interested reader, Section I.7 in online Appendix I deals with these issues in detail.

Fifth 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 because 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.

However, 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. For example, for SPECint2006 running on the i7 6700, which uses early restart and critical word first, there is more than one reference made to a block with an outstanding miss (1.23 references on average with a range from 0.5 to 3.0). We explore the performance of the i7 memory hierarchy in more detail in Section 2.6.

Sixth 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 because 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%–10% performance loss.

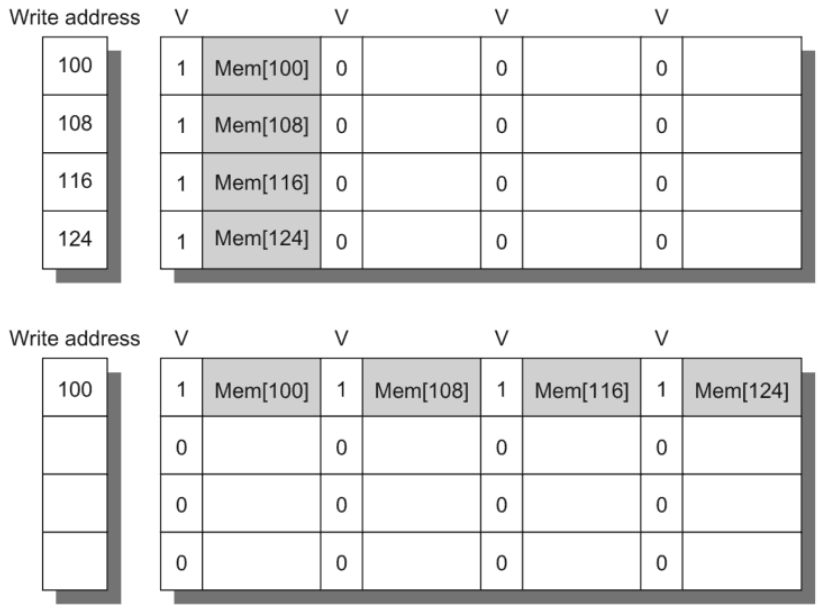


Figure 2.12 In this illustration of write merging, the write buffer on top does not use write merging while the write buffer on the bottom does. The four writes are merged into a single buffer entry with write merging; without it, the buffer is full even though three-fourths of each entry is wasted. 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 be used only for instructions that wrote multiple words at the same time.)

The optimization also reduces stalls because of the write buffer being full. Figure 2.12 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 fit exactly 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.

Seventh 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 is split between improvements in instruction misses and improvements in data misses. The optimizations presented next 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 because the array is laid out by rows), then the two pieces of the following code 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.

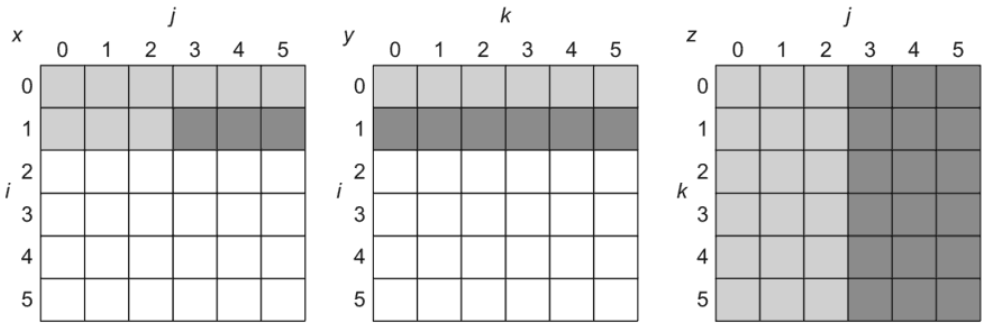


Figure 2.13 A snapshot of the three arrays `x`, `y`, and `z` when *N* = 6 and `i` = 1. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. The elements of `y` and `z` are read repeatedly to calculate new elements of `x`. The variables `i`, `j`, and `k` are shown along the rows or columns used to access the arrays.

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 following code example, 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 2.13 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.

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 2.14 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 an approximate factor of B. Therefore blocking exploits a combination of spatial and temporal locality, because y benefits from spatial locality and z benefits from temporal locality. Although our example uses a square block (BxB), we could also use a rectangular block, which would be necessary if the matrix were not square.

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.

Eighth 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

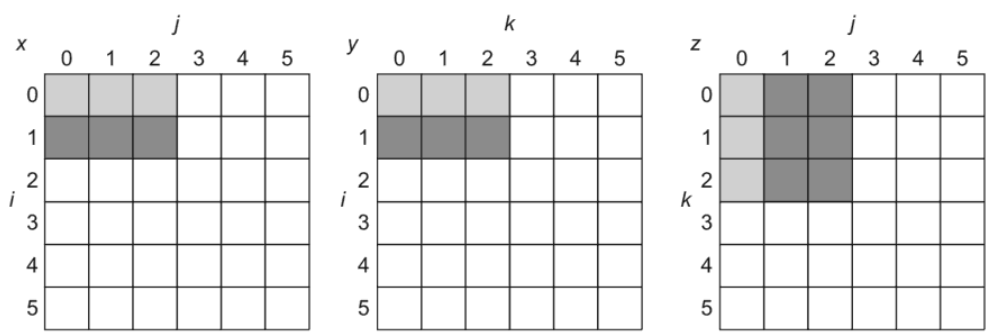


Figure 2.14 The age of accesses to the arrays x, y, and z when $B = 3$. Note that, in contrast to Figure 2.13, a smaller number of elements is accessed.

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 in 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%–70% of all misses from a processor with two 64 KiB 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 2.15 shows the overall performance improvement for a subset of SPEC2000 programs when hardware prefetching is turned on. Note that this figure

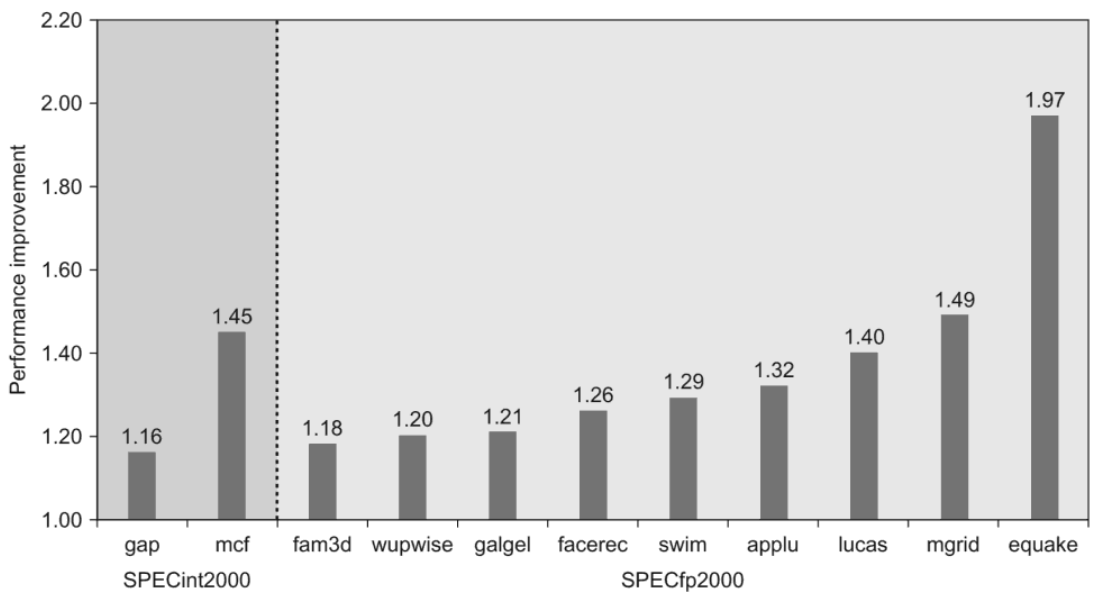


Figure 2.15 Speedup because of hardware prefetching on Intel Pentium 4 with hardware prefetching turned on for 2 of 12 SPECint2000 benchmarks and 9 of 14 SPECfp2000 benchmarks. Only the programs that benefit the most from prefetching are shown; prefetching speeds up the missing 15 SPECCPU benchmarks by less than 15% (Boggs et al., 2004).

includes only 2 of 12 integer programs, while it includes the majority of the SPECCPU floating-point programs. We will return to our evaluation of prefetching on the i7 in Section 2.6.

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.

Ninth 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* loads 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 because 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 following code, 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 KiB 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 because 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. Because `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 because 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 because of `b` will be for `b[j+1][0]` accesses when `i=0`, and also the first access to `b[j][0]` when `j=0`. Because `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]` … `a[i][106]`) and the end of `b` (`b[101][0]` … `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 preceding code 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]`, … , `b[6][0]` in the first loop
- 4 misses ([7/2]) for elements `a[0][0]`, `a[0][1]`, … , `a[0][6]` in the first loop (spatial locality reduces misses to 1 per 16-byte cache block)
- 4 misses ([7/2]) for elements `a[1][0]`, `a[1][1]`, … , `a[1][6]` in the second loop
- 4 misses ([7/2]) for elements `a[2][0]`, `a[2][1]`, … , `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 preceding example. 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. Because 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%–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.

Tenth Optimization: Using HBM to Extend the Memory Hierarchy

Because most general-purpose processors in servers will likely want more memory than can be packaged with HBM packaging, it has been proposed that the in-package DRAMs be used to build massive L4 caches, with upcoming technologies ranging from 128 MiB to 1 GiB and more, considerably more than current on-chip L3 caches. Using such large DRAM-based caches raises an issue: where do the tags reside? That depends on the number of tags. Suppose we were to use a 64B block size; then a 1 GiB L4 cache requires 96 MiB of tags—far more static memory than exists in the caches on the CPU. Increasing the block size to 4 KiB, yields a dramatically reduced tag store of 256 K entries or less than 1 MiB total storage, which is probably acceptable, given L3 caches of 4–16 MiB or more in next-generation, multicore processors. Such large block sizes, however, have two major problems.

First, the cache may be used inefficiently when content of many blocks are not needed; this is called the *fragmentation problem*, and it also occurs in virtual memory systems. Furthermore, transferring such large blocks is inefficient if much of the data is unused. Second, because of the large block size, the number of distinct blocks held in the DRAM cache is much lower, which can result in more misses, especially for conflict and consistency misses.

One partial solution to the first problem is to add *sublocking*. *Subblocking* allow parts of the block to be invalid, requiring that they be fetched on a miss. Subblocking, however, does nothing to address the second problem.

The tag storage is the major drawback for using a smaller block size. One possible solution for that difficulty is to store the tags for L4 in the HBM. At first glance this seems unworkable, because it requires two accesses to DRAM for each L4 access: one for the tags and one for the data itself. Because of the long access time for random DRAM accesses, typically 100 or more processor clock cycles, such an approach had been discarded. Loh and Hill (2011) proposed a clever solution to this problem: place the tags and the data in the same row in the HBM SDRAM. Although opening the row (and eventually closing it) takes a large amount of time, the CAS latency to access a different part of the row is about one-third the new row access time. Thus we can access the tag portion of the block first, and if it is a hit,

then use a column access to choose the correct word. Loh and Hill (L-H) have proposed organizing the L4 HBM cache so that each SDRAM row consists of a set of tags (at the head of the block) and 29 data segments, making a 29-way set associative cache. When L4 is accessed, the appropriate row is opened and the tags are read; a hit requires one more column access to get the matching data.

Qureshi and Loh (2012) proposed an improvement called an *alloy cache* that reduces the hit time. An *alloy cache* molds the tag and data together and uses a direct mapped cache structure. This allows the L4 access time to be reduced to a single HBM cycle by directly indexing the HBM cache and doing a burst transfer of both the tag and data. Figure 2.16 shows the hit latency for the alloy cache, the L-H scheme, and SRAM based tags. The alloy cache reduces hit time by more than a factor of 2 versus the L-H scheme, in return for an increase in the miss rate by a factor of 1.1–1.2. The choice of benchmarks is explained in the caption.

Unfortunately, in both schemes, misses require two full DRAM accesses: one to get the initial tag and a follow-on access to the main memory (which is even

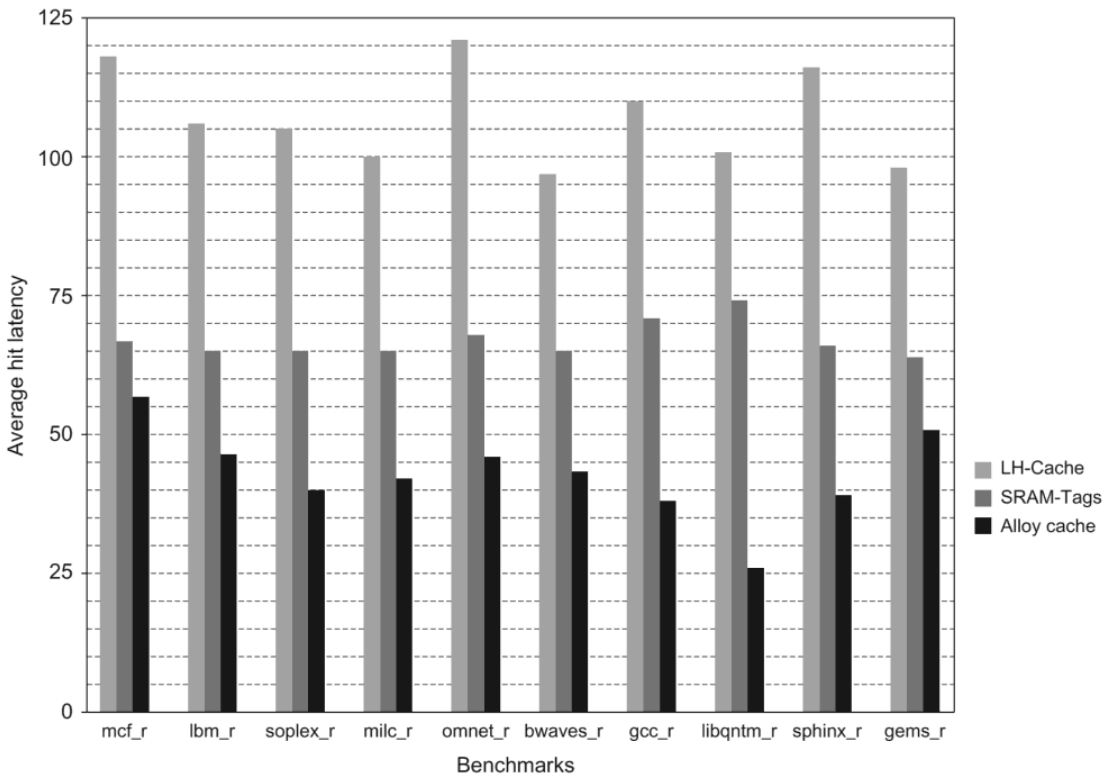


Figure 2.16 Average hit time latency in clock cycles for the L-H scheme, a currently-impractical scheme using SRAM for the tags, and the alloy cache organization. In the SRAM case, we assume the SRAM is accessible in the same time as L3 and that it is checked before L4 is accessed. The average hit latencies are 43 (alloy cache), 67 (SRAM tags), and 107 (L-H). The 10 SPECCPU2006 benchmarks used here are the most memory-intensive ones; each of them would run twice as fast if L3 were perfect.

slower). If we could speed up the miss detection, we could reduce the miss time. Two different solutions have been proposed to solve this problem: one uses a map that keeps track of the blocks in the cache (not the location of the block, just whether it is present); the other uses a memory access predictor that predicts likely misses using history prediction techniques, similar to those used for global branch prediction (see the next chapter). It appears that a small predictor can predict likely misses with high accuracy, leading to an overall lower miss penalty.

Figure 2.17 shows the speedup obtained on SPECrate for the memory-intensive benchmarks used in Figure 2.16. The alloy cache approach outperforms the LH scheme and even the impractical SRAM tags, because the combination of a fast access time for the miss predictor and good prediction results lead to a shorter time to predict a miss, and thus a lower miss penalty. The alloy cache performs close to the Ideal case, an L4 with perfect miss prediction and minimal hit time.

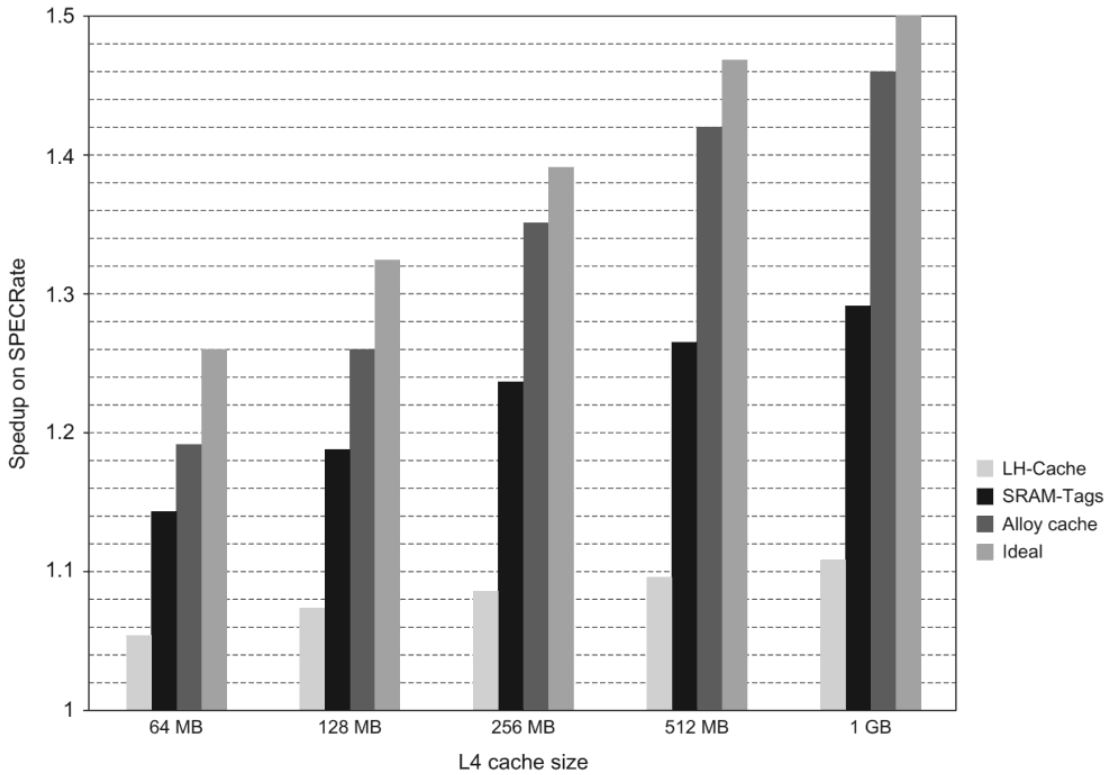


Figure 2.17 Performance speedup running the SPECrate benchmark for the LH scheme, an SRAM tag scheme, and an ideal L4 (Ideal); a speedup of 1 indicates no improvement with the L4 cache, and a speedup of 2 would be achievable if L4 were perfect and took no access time. The 10 memory-intensive benchmarks are used with each benchmark run eight times. The accompanying miss prediction scheme is used. The Ideal case assumes that only the 64-byte block requested in L4 needs to be accessed and transferred and that prediction accuracy for L4 is perfect (i.e., all misses are known at zero cost).

HBM is likely to have widespread use in a variety of different configurations, from containing the entire memory system for some high-performance, special-purpose systems to use as an L4 cache for larger server configurations.

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 2.18 summarizes these techniques and estimates the impact on complexity, with + meaning that the technique

Technique	Hit time	Band-width	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 & banked caches	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
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
HBM as additional level of cache		+/–	–	+	+	3	Depends on new packaging technology. Effects depend heavily on hit rate improvements

Figure 2.18 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.

improves the factor, – meaning it hurts that factor, and blank meaning it has no impact. Generally, no technique helps more than one category.

2.4 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).

Section B.4 in Appendix B describes the key concepts in virtual memory. Recall that virtual memory allows the physical memory to be treated as a cache of secondary storage (which may be either disk or solid state). Virtual memory moves pages between the two levels of the memory hierarchy, just as caches move blocks between levels. Likewise, TLBs act as caches on the page table, eliminating the need to do a memory access every time an address is translated. Virtual memory also provides separation between processes that share one physical memory but have separate virtual address spaces. Readers should ensure that they understand both functions of virtual memory before continuing.

In this section, we focus on additional issues in protection and privacy between processes sharing the same processor. Security and privacy are two of the most vexing challenges for information technology in 2017. Electronic burglaries, often involving lists of credit card numbers, are announced regularly, and it's widely believed that many more go unreported. Of course, such problems arise from programming errors that allow a cyberattack to access data it should be unable to access. Programming errors are a fact of life, and with modern complex software systems, they occur with significant regularity. Therefore both researchers and practitioners are looking for improved 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 by 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 TLB 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 section acts as a quick review; if it's too quick, please refer to the denoted Appendix B sections.

Multiprogramming, where several programs running concurrently share a computer, has 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 KiB, 16 KiB, or larger, 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. Because 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 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. Because 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 because of

- 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 data center or cloud; and
- 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 in order to allow software from the departing ISA to be used until it can be ported to the new ISA. Our focus here will be on 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 SPECCPU2006, 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. Therefore, 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 because 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 most newer VMMs support migration of a running VM to a different computer, either to

balance load or to evacuate from failing hardware. The rise of cloud computing has made the ability to swap out an entire VM to another physical processor increasingly useful.

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 directly change allocation of real system resources.

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 the previously mentioned paged virtual memory:

- 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.

Instruction Set Architecture Support for Virtual Machines

If VMs are planned for during the design of the ISA, it's relatively easy to reduce both 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.

However, because 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 of the original RISC architectures, although the latter had fewer issues than the 80x86 architecture. Recent additions to the x86 architecture have attempted to remedy the earlier shortcomings, and RISC V explicitly includes support for virtualization.

Because the VMM must ensure that the guest system interacts only 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.

Therefore, 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.

In the absence of such support, other measures must be taken. A VMM must take special precautions to locate all problematic instructions and ensure that they behave correctly when executed by a guest OS, thereby increasing the complexity of the VMM and reducing the performance of running the VM. Sections 2.5 and 2.7 give concrete examples of problematic instructions in the 80x86 architecture. One attractive extension allows the VM and the OS to operate at different privilege levels, each of which is distinct from the user level. By introducing an additional privilege level, some OS operations—e.g., those that exceed the permissions granted to a user program but do not require intervention by the VMM (because they cannot affect any other VM)—can execute directly without the overhead of trapping and invoking the VMM. The Xen design, which we examine shortly, makes use of three privilege levels.

Impact of Virtual Machines on Virtual Memory and I/O

Another challenge is virtualization of virtual memory, as each guest OS in every VM manages its own set of page tables. To make this work, the VMM separates the notions of *real* and *physical memory* (which are often treated synonymously) and makes real memory a separate, intermediate level between virtual memory and physical memory. (Some use the terms *virtual memory, physical memory*, and *machine memory* to name the same three levels.) The guest OS maps virtual memory to real memory via its page tables, and the VMM page tables map the guests' real memory to physical memory. The virtual memory architecture is specified either via page tables, as in IBM VM/370 and the 80x86, or via the TLB structure, as in many RISC architectures.

Rather than pay an extra level of indirection on every memory access, the VMM maintains a *shadow page table* that maps directly from the guest virtual address space to the physical address space of the hardware. By detecting all modifications to the guest's page table, the VMM can ensure that the shadow page table

entries being used by the hardware for translations correspond to those of the guest OS environment, with the exception of the correct physical pages substituted for the real pages in the guest tables. Therefore the VMM must trap any attempt by the guest OS to change its page table or to access the page table pointer. This is commonly done by write protecting the guest page tables and trapping any access to the page table pointer by a guest OS. As previously noted, the latter happens naturally if accessing the page table pointer is a privileged operation.

The IBM 370 architecture solved the page table problem in the 1970s with an additional level of indirection that is managed by the VMM. The guest OS keeps its page tables as before, so the shadow pages are unnecessary. AMD has implemented a similar scheme for its 80x86.

To virtualize the TLB in many RISC computers, the VMM manages the real TLB and has a copy of the contents of the TLB of each guest VM. To pull this off, any instructions that access the TLB must trap. TLBs with Process ID tags can support a mix of entries from different VMs and the VMM, thereby avoiding flushing of the TLB on a VM switch. Meanwhile, in the background, the VMM supports a mapping between the VMs' virtual Process IDs and the real Process IDs. Section L.7 of online Appendix L describes additional details.

The final portion of the architecture to virtualize is I/O. This is by far the most difficult part of system virtualization because of the increasing number of I/O devices attached to the computer *and* the increasing diversity of I/O device types. Another difficulty is the sharing of a real device among multiple VMs, and yet another comes from supporting the myriad of device drivers that are required, especially if different guest OSes are supported on the same VM system. The VM illusion can be maintained by giving each VM generic versions of each type of I/O device driver, and then leaving it to the VMM to handle real I/O.

The method for mapping a virtual-to-physical I/O device depends on the type of device. For example, physical disks are normally partitioned by the VMM to create virtual disks for guest VMs, and the VMM maintains the mapping of virtual tracks and sectors to the physical ones. Network interfaces are often shared between VMs in very short time slices, and the job of the VMM is to keep track of messages for the virtual network addresses to ensure that guest VMs receive only messages intended for them.

Extending the Instruction Set for Efficient Virtualization and Better Security

In the past 5–10 years, processor designers, including those at AMD and Intel (and to a lesser extent ARM), have introduced instruction set extensions to more efficiently support virtualization. Two primary areas of performance improvement have been in handling page tables and TLBs (the cornerstone of virtual memory) and in I/O, specifically handling interrupts and DMA. Virtual memory performance is enhanced by avoiding unnecessary TLB flushes and by using the nested page table mechanism, employed by IBM decades earlier, rather than a complete

set of shadow page tables (see Section L.7 in Appendix L). To improve I/O performance, architectural extensions are added that allow a device to directly use DMA to move data (eliminating a potential copy by the VMM) and allow device interrupts and commands to be handled by the guest OS directly. These extensions show significant performance gains in applications that are intensive either in their memory-management aspects or in the use of I/O.

With the broad adoption of public cloud systems for running critical applications, concerns have risen about security of data in such applications. Any malicious code that is able to access a higher privilege level than data that must be kept secure compromises the system. For example, if you are running a credit card processing application, you must be absolutely certain that malicious users cannot get access to the credit card numbers, even when they are using the same hardware and intentionally attack the OS or even the VMM. Through the use of virtualization, we can prevent accesses by an outside user to the data in a different VM, and this provides significant protection compared to a multiprogrammed environment. That might not be enough, however, if the attacker compromises the VMM or can find out information by observations in another VMM. For example, suppose the attacker penetrates the VMM; the attacker can then remap memory so as to access any portion of the data.

Alternatively, an attack might rely on a Trojan horse (see Appendix B) introduced into the code that can access the credit cards. Because the Trojan horse is running in the same VM as the credit card processing application, the Trojan horse only needs to exploit an OS flaw to gain access to the critical data. Most cyberattacks have used some form of Trojan horse, typically exploiting an OS flaw, that either has the effect of returning access to the attacker while leaving the CPU still in privilege mode or allows the attacker to upload and execute code as if it were part of the OS. In either case, the attacker obtains control of the CPU and, using the higher privilege mode, can proceed to access anything within the VM. Note that encryption alone does not prevent this attacker. If the data in memory is unencrypted, which is typical, then the attacker has access to all such data. Furthermore, if the attacker knows where the encryption key is stored, the attacker can freely access the key and then access any encrypted data.

More recently, Intel introduced a set of instruction set extensions, called the software guard extensions (SGX), to allow user programs to create *enclaves*, portions of code and data that are always encrypted and decrypted only on use and only with the key provided by the user code. Because the enclave is always encrypted, standard OS operations for virtual memory or I/O can access the enclave (e.g., to move a page) but cannot extract any information. For an enclave to work, all the code and all the data required must be part of the enclave. Although the topic of finer-grained protection has been around for decades, it has gotten little traction before because of the high overhead and because other solutions that are more efficient and less intrusive have been acceptable. The rise of cyberattacks and the amount of confidential information online have led to a reexamination of techniques for improving such fine-grained security. Like Intel's SGX, IBM and AMD's recent processors support on-the-fly encryption of memory.

An Example VMM: The Xen Virtual Machine

Early in the development of VMs, a number of inefficiencies became apparent. For example, a guest OS manages its virtual-to-real page mapping, but this mapping is ignored by the VMM, which performs the actual mapping to physical pages. In other words, a significant amount of wasted effort is expended just to keep the guest OS happy. To reduce such inefficiencies, VMM developers decided that it may be worthwhile to allow the guest OS to be aware that it is running on a VM. For example, a guest OS could assume a real memory as large as its virtual memory so that no memory management is required by the guest OS.

Allowing small modifications to the guest OS to simplify virtualization is referred to as *paravirtualization*, and the open source Xen VMM is a good example. The Xen VMM, which is used in Amazon's web services data centers, provides a guest OS with a virtual machine abstraction that is similar to the physical hardware, but drops many of the troublesome pieces. For example, to avoid flushing the TLB, Xen maps itself into the upper 64 MiB of the address space of each VM. Xen allows the guest OS to allocate pages, checking only to be sure the guest OS does not violate protection restrictions. To protect the guest OS from the user programs in the VM, Xen takes advantage of the four protection levels available in the 80x86. The Xen VMM runs at the highest privilege level (0), the guest OS runs at the next level (1), and the applications run at the lowest privilege level (3). Most OSes for the 80x 86 keep everything at privilege levels 0 or 3.

For subsetting to work properly, Xen modifies the guest OS to not use problematic portions of the architecture. For example, the port of Linux to Xen changes about 3000 lines, or about 1% of the 80x86-specific code. These changes, however, do not affect the application binary interfaces of the guest OS.

To simplify the I/O challenge of VMs, Xen assigned privileged virtual machines to each hardware I/O device. These special VMs are called *driver domains*. (Xen calls VMs "domains.") Driver domains run the physical device drivers, although interrupts are still handled by the VMM before being sent to the appropriate driver domain. Regular VMs, called *guest domains*, run simple virtual device drivers that must communicate with the physical device drivers in the driver domains over a channel to access the physical I/O hardware. Data are sent between guest and driver domains by page remapping.

2.5 Cross-Cutting Issues: The Design of Memory Hierarchies

This section describes four topics discussed in other chapters that are fundamental to memory hierarchies.

Protection, Virtualization, and Instruction Set Architecture

Protection is a joint effort of architecture and operating systems, but architects had to modify some awkward details of existing instruction set architectures when virtual memory became popular. For example, to support virtual memory in the IBM

370, architects had to change the successful IBM 360 instruction set architecture that had been announced just 6 years before. Similar adjustments are being made today to accommodate virtual machines.

For example, the 80x86 instruction POPF loads the flag registers from the top of the stack in memory. One of the flags is the Interrupt Enable (IE) flag. Until recent changes to support virtualization, running the POPF instruction in user mode, rather than trapping it, simply changed all the flags except IE. In system mode, it does change the IE flag. Because a guest OS runs in user mode inside a VM, this was a problem, as the OS would expect to see a changed IE. Extensions of the 80x86 architecture to support virtualization eliminated this problem.

Historically, IBM mainframe hardware and VMM took three steps to improve performance of virtual machines:

1. Reduce the cost of processor virtualization.
2. Reduce interrupt overhead cost due to the virtualization.
3. Reduce interrupt cost by steering interrupts to the proper VM without invoking VMM.

IBM is still the gold standard of virtual machine technology. For example, an IBM mainframe ran thousands of Linux VMs in 2000, while Xen ran 25 VMs in 2004 (Clark et al., 2004). Recent versions of Intel and AMD chipsets have added special instructions to support devices in a VM to mask interrupts at lower levels from each VM and to steer interrupts to the appropriate VM.

Autonomous Instruction Fetch Units

Many processors with out-of-order execution and even some with simply deep pipelines decouple the instruction fetch (and sometimes initial decode), using a separate instruction fetch unit (see Chapter 3). Typically, the instruction fetch unit accesses the instruction cache to fetch an entire block before decoding it into individual instructions; such a technique is particularly useful when the instruction length varies. Because the instruction cache is accessed in blocks, it no longer makes sense to compare miss rates to processors that access the instruction cache once per instruction. In addition, the instruction fetch unit may prefetch blocks into the L1 cache; these prefetches may generate additional misses, but may actually reduce the total miss penalty incurred. Many processors also include data prefetching, which may increase the data cache miss rate, even while decreasing the total data cache miss penalty.

Speculation and Memory Access

One of the major techniques used in advanced pipelines is speculation, whereby an instruction is tentatively executed before the processor knows whether it is really needed. Such techniques rely on branch prediction, which if incorrect requires that

the speculated instructions are flushed from the pipeline. There are two separate issues in a memory system supporting speculation: protection and performance. With speculation, the processor may generate memory references, which will never be used because the instructions were the result of incorrect speculation. Those references, if executed, could generate protection exceptions. Obviously, such faults should occur only if the instruction is actually executed. In the next chapter, we will see how such "speculative exceptions" are resolved. Because a speculative processor may generate accesses to both the instruction and data caches, and subsequently not use the results of those accesses, speculation may increase the cache miss rates. As with prefetching, however, such speculation may actually lower the total cache miss penalty. The use of speculation, like the use of prefetching, makes it misleading to compare miss rates to those seen in processors without speculation, even when the ISA and cache structures are otherwise identical.

Special Instruction Caches

One of the biggest challenges in superscalar processors is to supply the instruction bandwidth. For designs that translate the instructions into micro-operations, such as most recent Arm and i7 processors, instruction bandwidth demands and branch misprediction penalties can be reduced by keeping a small cache of recently translated instructions. We explore this technique in greater depth in the next chapter.

Coherency of Cached Data

Data can be found in memory and in the cache. As long as the processor is the sole component changing or reading the data and the cache stands between the processor and memory, there is little danger in the processor seeing the old or *stale* copy. As we will see, multiple processors and I/O devices raise the opportunity for copies to be inconsistent and to read the wrong copy.

The frequency of the cache coherency problem is different for multiprocessors than for I/O. Multiple data copies are a rare event for I/O—one to be avoided whenever possible—but a program running on multiple processors will *want* to have copies of the same data in several caches. Performance of a multiprocessor program depends on the performance of the system when sharing data.

The *I/O cache coherency* question is this: where does the I/O occur in the computer—between the I/O device and the cache or between the I/O device and main memory? If input puts data into the cache and output reads data from the cache, both I/O and the processor see the same data. The difficulty in this approach is that it interferes with the processor and can cause the processor to stall for I/O. Input may also interfere with the cache by displacing some information with new data that are unlikely to be accessed soon.

The goal for the I/O system in a computer with a cache is to prevent the stale data problem while interfering as little as possible. Many systems therefore prefer that I/O occur directly to main memory, with main memory acting as an I/O buffer. If a write-through cache were used, then memory would have an up-to-date copy of the information, and there would be no stale data issue for output. (This benefit is a reason processors used write through.) However, today write through is usually found only in first-level data caches backed by an L2 cache that uses write back.

Input requires some extra work. The software solution is to guarantee that no blocks of the input buffer are in the cache. A page containing the buffer can be marked as noncachable, and the operating system can always input to such a page. Alternatively, the operating system can flush the buffer addresses from the cache before the input occurs. A hardware solution is to check the I/O addresses on input to see if they are in the cache. If there is a match of I/O addresses in the cache, the cache entries are invalidated to avoid stale data. All of these approaches can also be used for output with write-back caches.

Processor cache coherency is a critical subject in the age of multicore processors, and we will examine it in detail in Chapter 5.

2.6 Putting It All Together: Memory Hierarchies in the ARM Cortex-A53 and Intel Core i7 6700

This section reveals the ARM Cortex-A53 (hereafter called the A53) and Intel Core i76700 (hereafter called i7) memory hierarchies and shows the performance of their components on a set of single-threaded benchmarks. We examine the Cortex-A53 first because it has a simpler memory system; we go into more detail for the i7, tracing out a memory reference in detail. This section presumes that readers are familiar with the organization of a two-level cache hierarchy using virtually indexed caches. The basics of such a memory system are explained in detail in Appendix B, and readers who are uncertain of the organization of such a system are strongly advised to review the Opteron example in Appendix B. Once they understand the organization of the Opteron, the brief explanation of the A53 system, which is similar, will be easy to follow.

The ARM Cortex-A53

The Cortex-A53 is a configurable core that supports the ARMv8A instruction set architecture, which includes both 32-bit and 64-bit modes. The Cortex-A53 is delivered as an IP (intellectual property) core. IP cores are the dominant form of technology delivery in the embedded, PMD, and related markets; billions of ARM and MIPS processors have been created from these IP cores. Note that IP cores are different from the cores in the Intel i7 or AMD Athlon multicores. An IP core (which may itself be a multicore) is designed to be incorporated with other logic (thus it is the core of a chip), including application-specific processors

(such as an encoder or decoder for video), I/O interfaces, and memory interfaces, and then fabricated to yield a processor optimized for a particular application. For example, the Cortex-A53 IP core is used in a variety of tablets and smartphones; it is designed to be highly energy-efficient, a key criteria in battery-based PMDs. The A53 core is capable of being configured with multiple cores per chip for use in high-end PMDs; our discussion here focuses on a single core.

Generally, IP cores come in two flavors. *Hard cores* are optimized for a particular semiconductor vendor and are black boxes with external (but still on-chip) interfaces. Hard cores typically allow parametrization only of logic outside the core, such as L2 cache sizes, and the IP core cannot be modified. *Soft cores* are usually delivered in a form that uses a standard library of logic elements. A soft core can be compiled for different semiconductor vendors and can also be modified, although extensive modifications are very difficult because of the complexity of modern-day IP cores. In general, hard cores provide higher performance and smaller die area, while soft cores allow retargeting to other vendors and can be more easily modified.

The Cortex-A53 can issue two instructions per clock at clock rates up to 1.3 GHz. It supports both a two-level TLB and a two-level cache; Figure 2.19 summarizes the organization of the memory hierarchy. The critical term is returned first, and the processor can continue while the miss completes; a memory system with up to four banks can be supported. For a D-cache of 32 KiB and a page size of 4 KiB, each physical page could map to two different cache addresses; such aliases are avoided by hardware detection on a miss as in Section B.3 of Appendix B. Figure 2.20 shows how the 32-bit virtual address is used to index the TLB and the caches, assuming 32 KiB primary caches and a 1 MiB secondary cache with 16 KiB page size.

Structure	Size	Organization	Typical miss penalty (clock cycles)
Instruction MicroTLB	10 entries	Fully associative	2
Data MicroTLB	10 entries	Fully associative	2
L2 Unified TLB	512 entries	4-way set associative	20
L1 Instruction cache	8–64 KiB	2-way set associative; 64-byte block	13
L1 Data cache	8–64 KiB	2-way set associative; 64-byte block	13
L2 Unified cache	128 KiB to 2 MiB	16-way set associative; LRU	124

Figure 2.19 The memory hierarchy of the Cortex A53 includes multilevel TLBs and caches. A page map cache keeps track of the location of a physical page for a set of virtual pages; it reduces the L2 TLB miss penalty. The L1 caches are virtually indexed and physically tagged; both the L1 D cache and L2 use a write-back policy defaulting to allocate on write. Replacement policy is LRU approximation in all the caches. Miss penalties to L2 are higher if both a MicroTLB and L1 miss occur. The L2 to main memory bus is 64–128 bits wide, and the miss penalty is larger for the narrow bus.

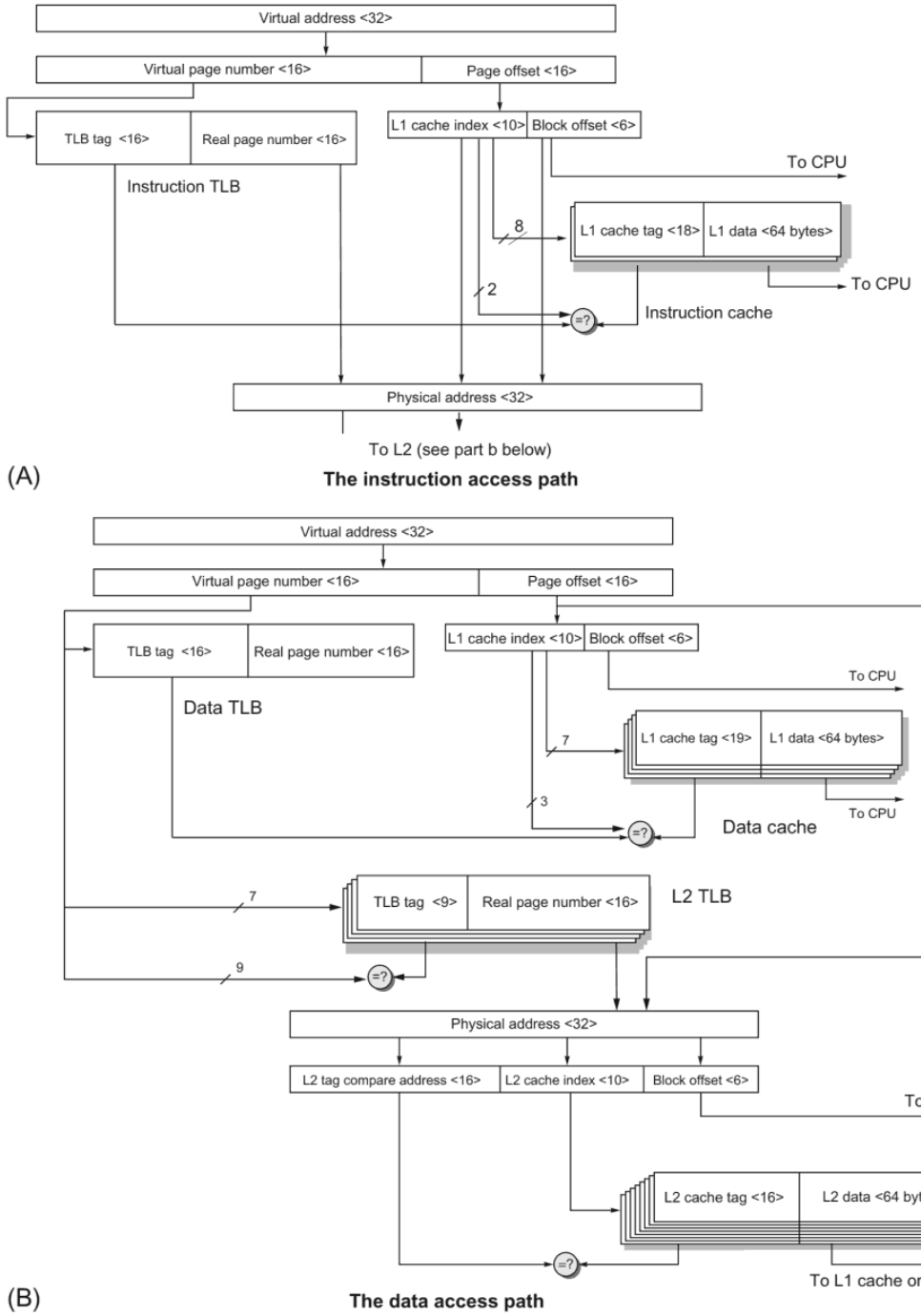


Figure 2.20 The virtual address, physical and data blocks for the ARM Cortex-A53 caches and TLBs, assuming 32-bit addresses. The top half (A) shows the instruction access; the bottom half (B) shows the data access, including L2. The TLB (instruction or data) is fully associative each with 10 entries, using a 64 KiB page in this example. The L1 I-cache is two-way set associative, with 64-byte blocks and 32 KiB capacity; the L1 D-cache is 32 KiB, four-way set associative, and 64-byte blocks. The L2 TLB is 512 entries and four-way set associative. The L2 cache is 16-way set associative with 64-byte blocks and 128 cKiB to 2 MiB capacity; a 1 MiB L2 is shown. This figure doesn't show the valid bits and protection bits for the caches and TLB.

Performance of the Cortex-A53 Memory Hierarchy

The memory hierarchy of the Cortex-A8 was measured with 32 KiB primary caches and a 1 MiB L2 cache running the SPECInt2006 benchmarks. The instruction cache miss rates for these SPECInt2006 are very small even for just the L1: close to zero for most and under 1% for all of them. This low rate probably results from the computationally intensive nature of the SPECCPU programs and the two-way set associative cache that eliminates most conflict misses.

Figure 2.21 shows the data cache results, which have significant L1 and L2 miss rates. The L1 rate varies by a factor of 75, from 0.5% to 37.3% with a median miss rate of 2.4%. The global L2 miss rate varies by a factor of 180, from 0.05% to 9.0% with a median of 0.3%. MCF, which is known as a cache buster, sets the upper bound and significantly affects the mean. Remember that the L2 global miss rate is significantly lower than the L2 local miss rate; for example, the median L2 stand-alone miss rate is 15.1% versus the global miss rate of 0.3%.

Using these miss penalties in Figure 2.19, Figure 2.22 shows the average penalty per data access. Although the L1 miss rates are about seven times higher than the L2 miss rate, the L2 penalty is 9.5 times as high, leading to L2 misses slightly dominating for the benchmarks that stress the memory system. In the next chapter, we will examine the impact of the cache misses on overall CPI.

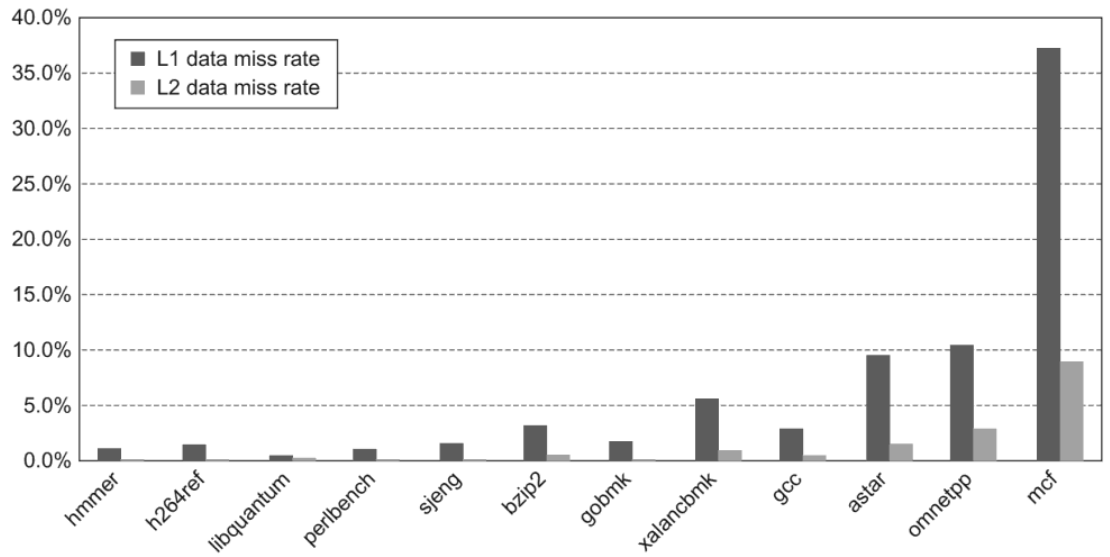


Figure 2.21 The data miss rate for ARM with a 32 KiB L1 and the global data miss rate for a 1 MiB L2 using the SPECInt2006 benchmarks are significantly affected by the applications. Applications with larger memory footprints tend to have higher miss rates in both L1 and L2. Note that the L2 rate is the global miss rate that is counting all references, including those that hit in L1. MCF is known as a cache buster.

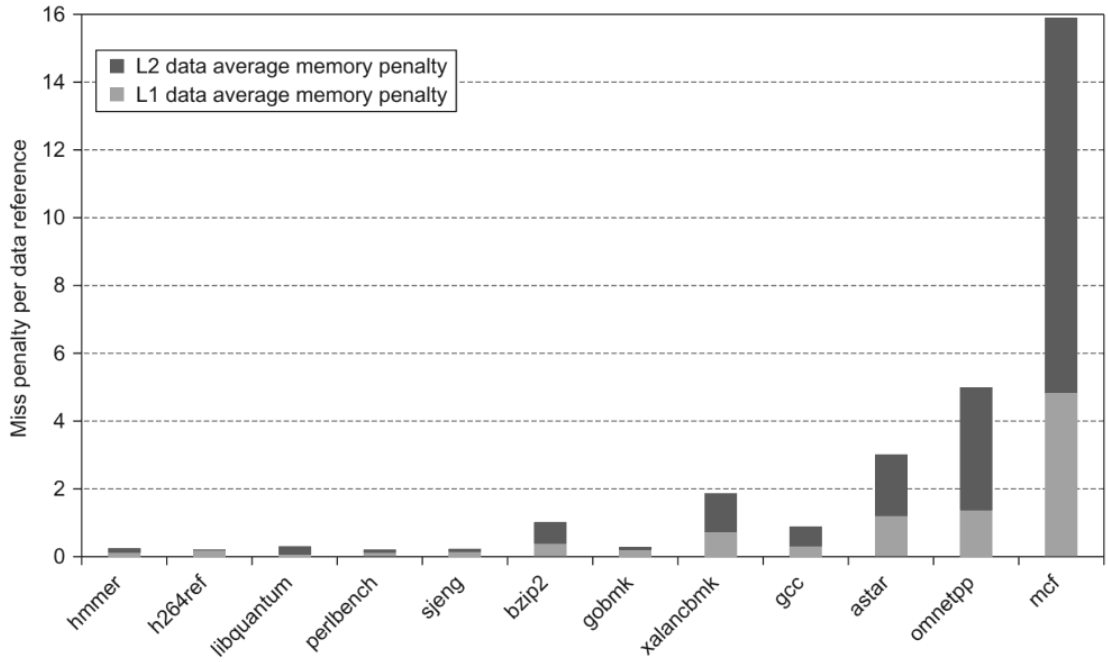


Figure 2.22 The average memory access penalty per data memory reference coming from L1 and L2 is shown for the A53 processor when running SPECInt2006. Although the miss rates for L1 are significantly higher, the L2 miss penalty, which is more than five times higher, means that the L2 misses can contribute significantly.

The Intel Core i7 6700

The i7 supports the x86-64 instruction set architecture, a 64-bit extension of the 80x86 architecture. The i7 is an out-of-order execution processor that includes four cores. In this chapter, we focus on the memory system design and performance from the viewpoint of a single core. The system performance of multiprocessor designs, including the i7 multicore, is examined in detail in Chapter 5.

Each core in an i7 can execute up to four 80x86 instructions per clock cycle, using a multiple issue, dynamically scheduled, 16-stage pipeline, which we describe in detail in Chapter 3. The i7 can also support up to two simultaneous threads per processor, using a technique called simultaneous multithreading, described in Chapter 4. In 2017 the fastest i7 had a clock rate of 4.0 GHz (in Turbo Boost mode), which yielded a peak instruction execution rate of 16 billion instructions per second, or 64 billion instructions per second for the four-core design. Of course, there is a big gap between peak and sustained performance, as we will see over the next few chapters.

The i7 can support up to three memory channels, each consisting of a separate set of DIMMs, and each of which can transfer in parallel. Using DDR3-1066 (DIMM PC8500), the i7 has a peak memory bandwidth of just over 25 GB/s.

i7 uses 48-bit virtual addresses and 36-bit physical addresses, yielding a maximum physical memory of 36 GiB. Memory management is handled with a two-level TLB (see Appendix B, Section B.4), summarized in Figure 2.23.

Figure 2.24 summarizes the i7's three-level cache hierarchy. The first-level caches are virtually indexed and physically tagged (see Appendix B, Section B.3), while the L2 and L3 caches are physically indexed. Some versions of the i7 6700 will support a fourth-level cache using HBM packaging.

Figure 2.25 is labeled with the steps of an access to the memory hierarchy. First, the PC is sent to the instruction cache. The instruction cache index is

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{32\text{K}}{64 \times 8} = 64 = 2^6$$

Characteristic	Instruction TLB	Data DLB	Second-level TLB
Entries	128	64	1536
Associativity	8-way	4-way	12-way
Replacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Access latency	1 cycle	1 cycle	8 cycles
Miss	9 cycles	9 cycles	Hundreds of cycles to access page table

Figure 2.23 Characteristics of the i7's TLB structure, which has separate first-level instruction and data TLBs, both backed by a joint second-level TLB. The first-level TLBs support the standard 4 KiB page size, as well as having a limited number of entries of large 2–4 MiB pages; only 4 KiB pages are supported in the second-level TLB. The i7 has the ability to handle two L2 TLB misses in parallel. See Section L.3 of online Appendix L for more discussion of multilevel TLBs and support for multiple page sizes.

Characteristic	L1	L2	L3
Size	32 KiB I/32 KiB D	256 KiB	2 MiB per core
Associativity	both 8-way	4-way	16-way
Access latency	4 cycles, pipelined	12 cycles	44 cycles
Replacement scheme	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU but with an ordered selection algorithm

Figure 2.24 Characteristics of the three-level cache hierarchy in the i7. All three caches use write back and a block size of 64 bytes. The L1 and L2 caches are separate for each core, whereas the L3 cache is shared among the cores on a chip and is a total of 2 MiB per core. All three caches are nonblocking and allow multiple outstanding writes. A merging write buffer is used for the L1 cache, which holds data in the event that the line is not present in L1 when it is written. (That is, an L1 write miss does not cause the line to be allocated.) L3 is inclusive of L1 and L2; we explore this property in further detail when we explain multiprocessor caches. Replacement is by a variant on pseudo-LRU; in the case of L3, the block replaced is always the lowest numbered way whose access bit is off. This is not quite random but is easy to compute.

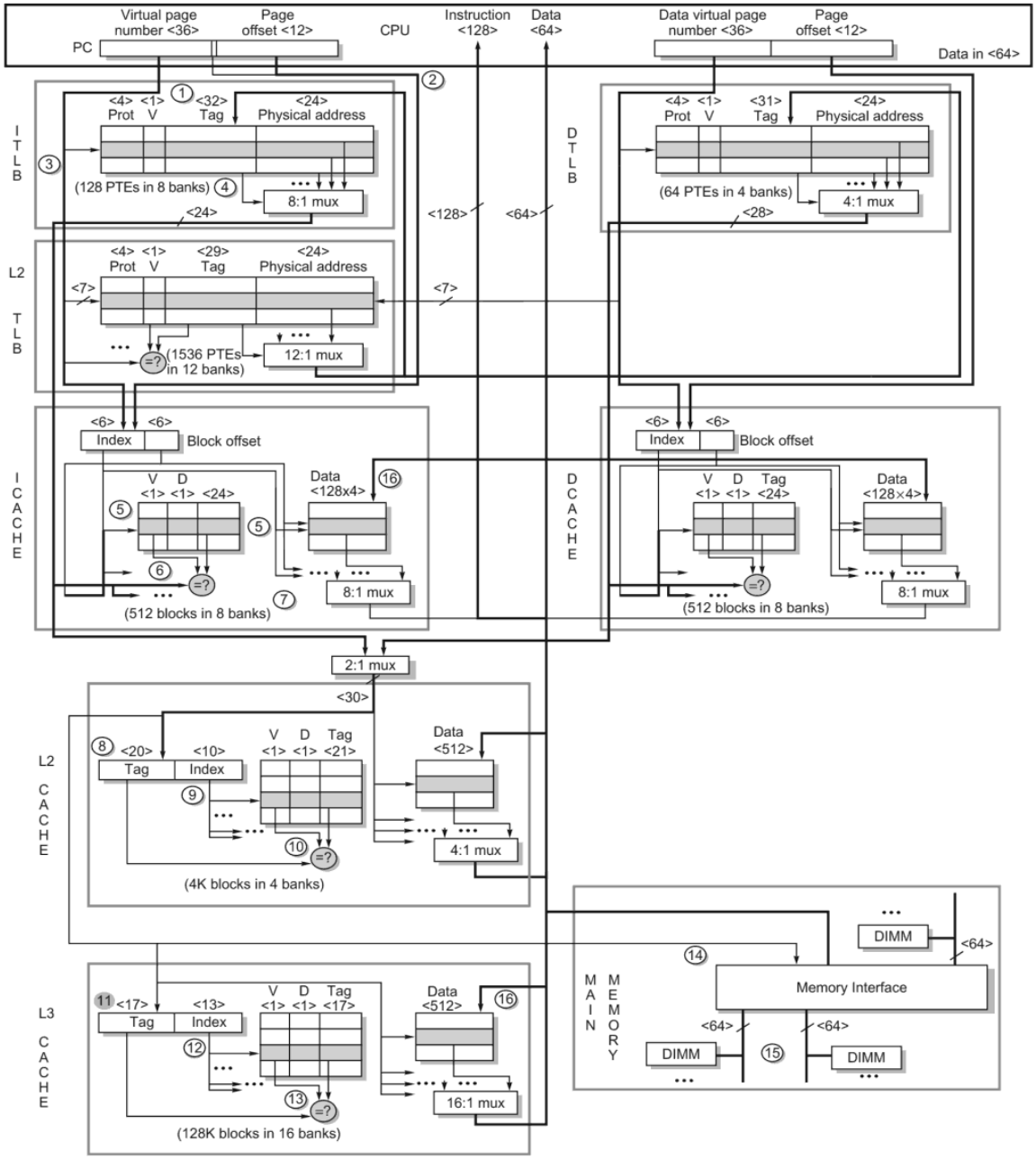


Figure 2.25 The Intel i7 memory hierarchy and the steps in both instruction and data access. We show only reads. Writes are similar, except that misses are handled by simply placing the data in a write buffer, because the L1 cache is not write-allocated.

or 6 bits. The page frame of the instruction's address (36 = 48 – 12 bits) is sent to the instruction TLB (step 1). At the same time, the 12-bit page offset from the virtual address is sent to the instruction cache (step 2). Notice that for the eight-way associative instruction cache, 12 bits are needed for the cache address: 6 bits to index the cache plus 6 bits of block offset for the 64-byte block, so no aliases are possible. The previous versions of the i7 used a four-way set associative I-cache, meaning that a block corresponding to a virtual address could actually be in two different places in the cache, because the corresponding physical address could have either a 0 or 1 in this location. For instructions this did not pose a problem because even if an instruction appeared in the cache in two different locations, the two versions must be the same. If such duplication, or aliasing, of data is allowed, the cache must be checked when the page map is changed, which is an infrequent event. Note that a very simple use of page coloring (see Appendix B, Section B.3) can eliminate the possibility of these aliases. If even-address virtual pages are mapped to even-address physical pages (and the same for odd pages), then these aliases can never occur because the low-order bit in the virtual and physical page number will be identical.

The instruction TLB is accessed to find a match between the address and a valid page table entry (PTE) (steps 3 and 4). In addition to translating the address, the TLB checks to see if the PTE demands that this access result in an exception because of an access violation.

An instruction TLB miss first goes to the L2 TLB, which contains 1536 PTEs of 4 KiB page sizes and is 12-way set associative. It takes 8 clock cycles to load the L1 TLB from the L2 TLB, which leads to the 9-cycle miss penalty including the initial clock cycle to access the L1 TLB. If the L2 TLB misses, a hardware algorithm is used to walk the page table and update the TLB entry. Sections L.5 and L.6 of online Appendix L describe page table walkers and page structure caches. In the worst case, the page is not in memory, and the operating system gets the page from secondary storage. Because millions of instructions could execute during a page fault, the operating system will swap in another process if one is waiting to run. Otherwise, if there is no TLB exception, the instruction cache access continues.

The index field of the address is sent to all eight banks of the instruction cache (step 5). The instruction cache tag is 36 bits – 6 bits (index) – 6 bits (block offset), or 24 bits. The four tags and valid bits are compared to the physical page frame from the instruction TLB (step 6). Because the i7 expects 16 bytes each instruction fetch, an additional 2 bits are used from the 6-bit block offset to select the appropriate 16 bytes. Therefore 6 + 2 or 8 bits are used to send 16 bytes of instructions to the processor. The L1 cache is pipelined, and the latency of a hit is 4 clock cycles (step 7). A miss goes to the second-level cache.

As mentioned earlier, the instruction cache is virtually addressed and physically tagged. Because the second-level caches are physically addressed, the physical page address from the TLB is composed with the page offset to make an address to access the L2 cache. The L2 index is

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{256\text{K}}{64 \times 4} = 1024 = 2^{10}$$

so the 30-bit block address (36-bit physical address − 6-bit block offset) is divided into a 20-bit tag and a 10-bit index (step 8). Once again, the index and tag are sent to the four banks of the unified L2 cache (step 9), which are compared in parallel. If one matches and is valid (step 10), it returns the block in sequential order after the initial 12-cycle latency at a rate of 8 bytes per clock cycle.

If the L2 cache misses, the L3 cache is accessed. For a four-core i7, which has an 8 MiB L3, the index size is

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{8\text{M}}{64 \times 16} = 8192 = 2^{13}$$

The 13-bit index (step 11) is sent to all 16 banks of the L3 (step 12). The L3 tag, which is $36 - (13 + 6) = 17$ bits, is compared against the physical address from the TLB (step 13). If a hit occurs, the block is returned after an initial latency of 42 clock cycles, at a rate of 16 bytes per clock and placed into both L1 and L3. If L3 misses, a memory access is initiated.

If the instruction is not found in the L3 cache, the on-chip memory controller must get the block from main memory. The i7 has three 64-bit memory channels that can act as one 192-bit channel, because there is only one memory controller and the same address is sent on both channels (step 14). Wide transfers happen when both channels have identical DIMMs. Each channel supports up to four DDR DIMMs (step 15). When the data return they are placed into L3 and L1 (step 16) because L3 is inclusive.

The total latency of the instruction miss that is serviced by main memory is approximately 42 processor cycles to determine that an L3 miss has occurred, plus the DRAM latency for the critical instructions. For a single-bank DDR4-2400 SDRAM and 4.0 GHz CPU, the DRAM latency is about 40 ns or 160 clock cycles to the first 16 bytes, leading to a total miss penalty of about 200 clock cycles. The memory controller fills the remainder of the 64-byte cache block at a rate of 16 bytes per I/O bus clock cycle, which takes another 5 ns or 20 clock cycles.

Because the second-level cache is a write-back cache, any miss can lead to an old block being written back to memory. The i7 has a 10-entry merging write buffer that writes back dirty cache lines when the next level in the cache is unused for a read. The write buffer is checked on a miss to see if the cache line exists in the buffer; if so, the miss is filled from the buffer. A similar buffer is used between the L1 and L2 caches. If this initial instruction is a load, the data address is sent to the data cache and data TLBs, acting very much like an instruction cache access.

Suppose the instruction is a store instead of a load. When the store issues, it does a data cache lookup just like a load. A miss causes the block to be placed in a write buffer because the L1 cache does not allocate the block on a write miss. On a hit, the store does not update the L1 (or L2) cache until later, after it is known to be nonspeculative. During this time, the store resides in a load-store queue, part of the out-of-order control mechanism of the processor.

The I7 also supports prefetching for L1 and L2 from the next level in the hierarchy. In most cases, the prefetched line is simply the next block in the cache. By prefetching only for L1 and L2, high-cost unnecessary fetches to memory are avoided.

Performance of the i7 memory system

We evaluate the performance of the i7 cache structure using the SPECint2006 benchmarks. The data in this section were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University. Their analysis is based on earlier work (see Prakash and Peng, 2008).

The complexity of the i7 pipeline, with its use of an autonomous instruction fetch unit, speculation, and both instruction and data prefetch, makes it hard to compare cache performance against simpler processors. As mentioned on page 110, processors that use prefetch can generate cache accesses independent of the memory accesses performed by the program. A cache access that is generated because of an actual instruction access or data access is sometimes called a *demand access* to distinguish it from a *prefetch access*. Demand accesses can come from both speculative instruction fetches and speculative data accesses, some of which are subsequently canceled (see Chapter 3 for a detailed description of speculation and instruction graduation). A speculative processor generates at least as many misses as an in-order nonspeculative processor, and typically more. In addition to demand misses, there are prefetch misses for both instructions and data.

The i7's instruction fetch unit attempts to fetch 16 bytes every cycle, which complicates comparing instruction cache miss rates because multiple instructions are fetched every cycle (roughly 4.5 on average). In fact, the entire 64-byte cache line is read and subsequent 16-byte fetches do not require additional accesses. Thus misses are tracked only on the basis of 64-byte blocks. The 32 KiB, eight-way set associative instruction cache leads to a very low instruction miss rate for the SPECint2006 programs. If, for simplicity, we measure the miss rate of SPECint2006 as the number of misses for a 64-byte block divided by the number of instructions that complete, the miss rates are all under 1% except for one benchmark (XALANCBMK), which has a 2.9% miss rate. Because a 64-byte block typically contains 16–20 instructions, the effective miss rate per instruction is much lower, depending on the degree of spatial locality in the instruction stream.

The frequency at which the instruction fetch unit is stalled waiting for the I-cache misses is similarly small (as a percentage of total cycles) increasing to 2% for two benchmarks and 12% for XALANCBMK, which has the highest I-cache miss rate. In the next chapter, we will see how stalls in the IFU contribute to overall reductions in pipeline throughput in the i7.

The L1 data cache is more interesting and even trickier to evaluate because in addition to the effects of prefetching and speculation, the L1 data cache is not write-allocated, and writes to cache blocks that are not present are not treated as misses. For this reason, we focus only on memory reads. The performance monitor measurements in the i7 separate out prefetch accesses from demand accesses, but only keep demand accesses for those instructions that graduate. The effect of speculative instructions that do not graduate is not negligible, although pipeline effects probably dominate secondary cache effects caused by speculation; we will return to the issue in the next chapter.

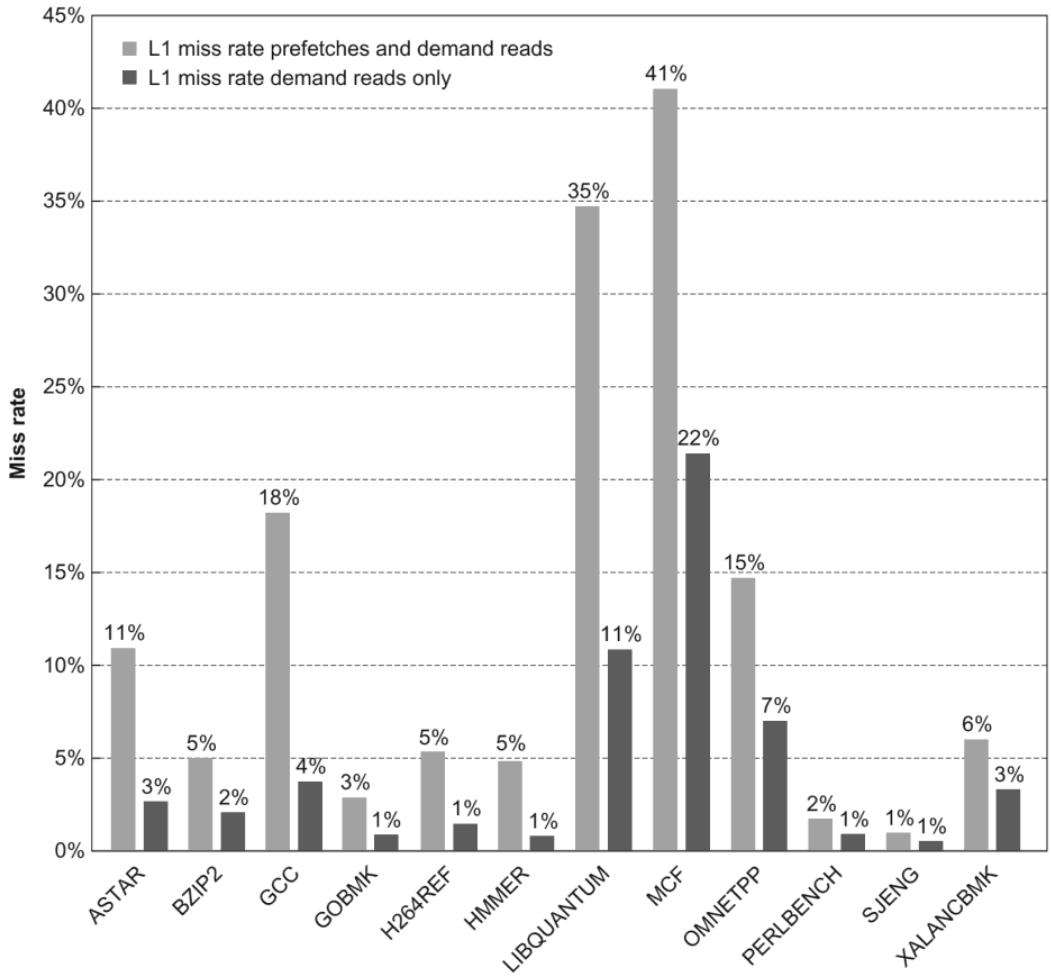


Figure 2.26 The L1 data cache miss rate for the SPECint2006 benchmarks is shown in two ways relative to the demand L1 reads: one including both demand and prefetch accesses and one including only demand accesses. The i7 separates out L1 misses for a block not present in the cache and L1 misses for a block already outstanding that is being prefetched from L2; we treat the latter group as hits because they would hit in a blocking cache. These data, like the rest in this section, were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University, based on earlier studies of the Intel Core Duo and other processors (see Peng et al., 2008).

To address these issues, while keeping the amount of data reasonable, Figure 2.26 shows the L1 data cache misses in two ways:

1. The L1 miss rate relative to demand references given by the L1 miss rate including prefetches and speculative loads/L1 demand read references for those instructions that graduate.

2. The demand miss rate given by L1 demand misses/L1 demand read references, both measurements only for instructions that graduate.

On average, the miss rate including prefetches is 2.8 times as high as the demand-only miss rate. Comparing this data to that from the earlier i7 920, which had the same size L1, we see that the miss rate including prefetches is higher on the newer i7, but the number of demand misses, which are more likely to cause a stall, are usually fewer.

To understand the effectiveness of the aggressive prefetch mechanisms in the i7, let's look at some measurements of prefetching. Figure 2.27 shows both the fraction of L2 requests that are prefetches versus demand requests and the prefetch miss rate. The data are probably astonishing at first glance: there are roughly 1.5 times as many prefetches as there are L2 demand requests, which come directly from L1 misses. Furthermore, the prefetch miss rate is amazingly high, with an average miss rate of 58%. Although the prefetch ratio varies considerably, the prefetch miss rate is always significant. At first glance, you might conclude that the designers made a mistake: they are prefetching too much, and the miss rate is too high. Notice, however, that the benchmarks with the higher prefetch ratios (ASTAR, BZIP2, HMMER, LIBQUANTUM, and OMNETPP) also show the greatest gap between the prefetch miss rate and the demand miss rate, more than a factor of 2 in each case. The aggressive prefetching is trading prefetch misses, which occur earlier, for demand misses, which occur later; and as a result, a pipeline stall is less likely to occur due to the prefetching.

Similarly, consider the high prefetch miss rate. Suppose that the majority of the prefetches are actually useful (this is hard to measure because it involves tracking individual cache blocks), then a prefetch miss indicates a likely L2 cache miss in the future. Uncovering and handling the miss earlier via the prefetch is likely to reduce the stall cycles. Performance analysis of speculative superscalars, like the i7, has shown that cache misses tend to be the primary cause of pipeline stalls, because it is hard to keep the processor going, especially for longer running L2 and L3 misses. The Intel designers could not easily increase the size of the caches without incurring both energy and cycle time impacts; thus the use of aggressive prefetching to try to lower effective cache miss penalties is an interesting alternative approach.

With the combination of the L1 demand misses and prefetches going to L2, roughly 17% of the loads generate an L2 request. Analyzing L2 performance requires including the effects of writes (because L2 is write-allocated), as well as the prefetch hit rate and the demand hit rate. Figure 2.28 shows the miss rates of the L2 caches for demand and prefetch accesses, both versus the number of L1 references (reads and writes). As with L1, prefetches are a significant contributor, generating 75% of the L2 misses. Comparing the L2 demand miss rate with that of earlier i7 implementations (again with the same L2 size) shows that the i7 6700 has a lower L2 demand miss rate by an approximate factor of 2, which may well justify the higher prefetch miss rate.

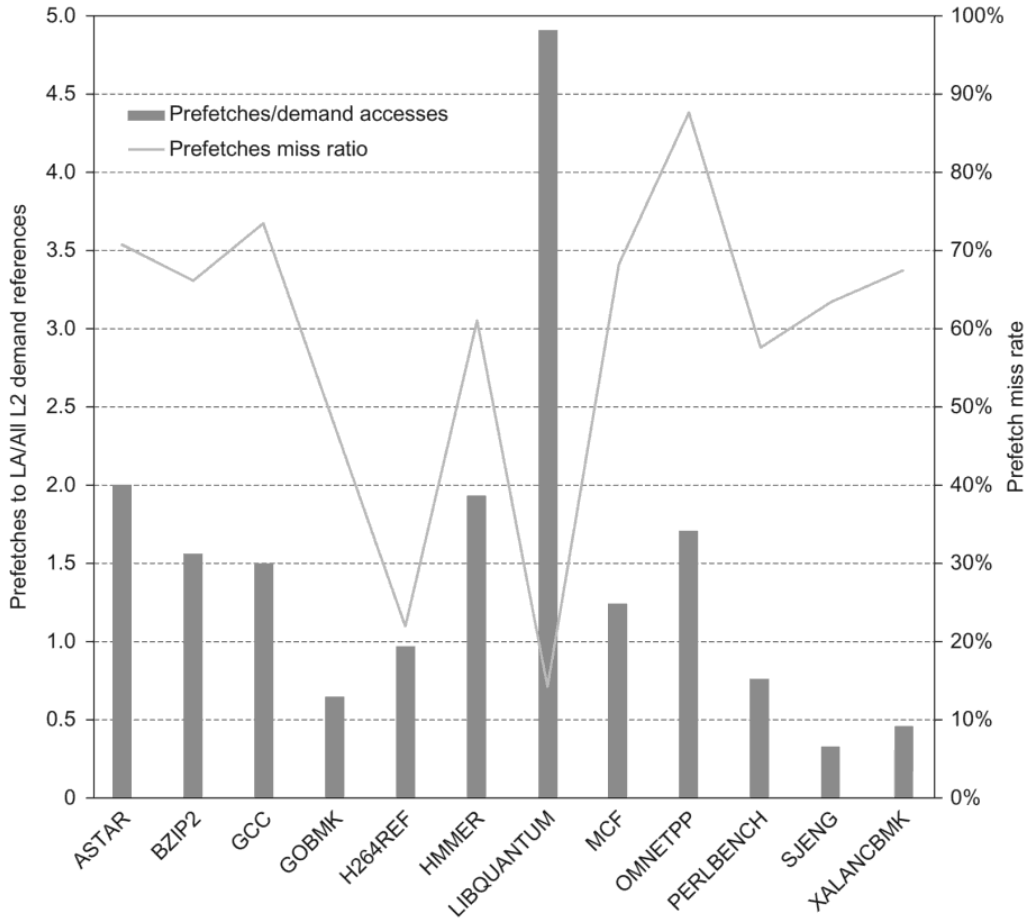


Figure 2.27 The fraction of L2 requests that are prefetches is shown via the columns and the left axis. The right axis and the line shows the prefetch hit rate. These data, like the rest in this section, were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University, based on earlier studies of the Intel Core Duo and other processors (see Peng et al., 2008).

Because the cost for a miss to memory is over 100 cycles and the average data miss rate in L2 combining both prefetch and demand misses is over 7%, L3 is obviously critical. Without L3 and assuming that about one-third of the instructions are loads or stores, L2 cache misses could add over two cycles per instruction to the CPI! Obviously, prefetching past L2 would make no sense without an L3.

In comparison, the average L3 data miss rate of 0.5% is still significant but less than one-third of the L2 demand miss rate and 10 times less than the L1 demand miss rate. Only in two benchmarks (OMNETPP and MCF) is the L3 miss rate

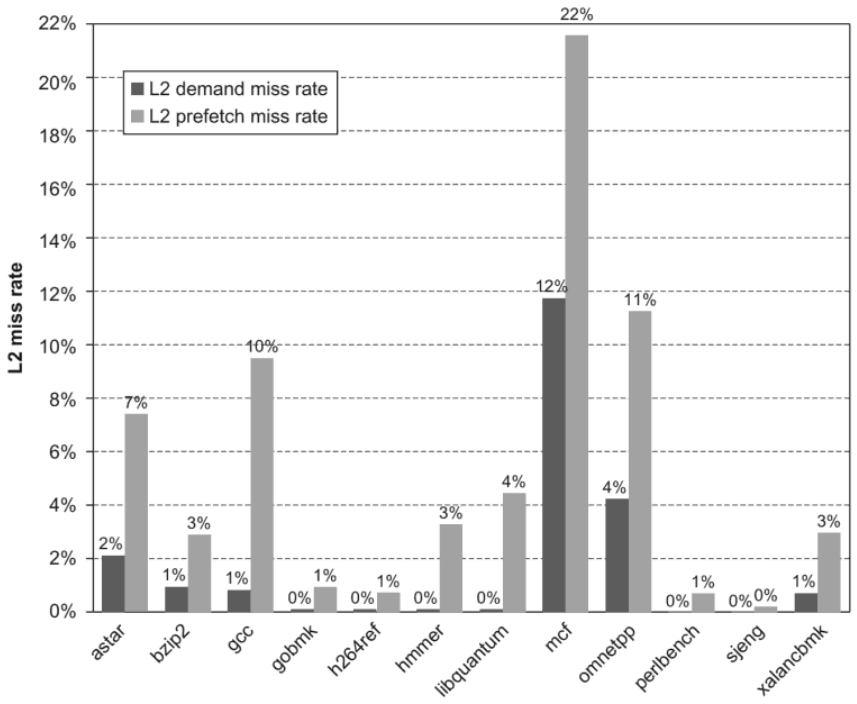


Figure 2.28 The L2 demand miss rate and prefetch miss rate, both shown relative to all the references to L1, which also includes prefetches, speculative loads that do not complete, and program-generated loads and stores (demand references). These data, like the rest in this section, were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University.

above 0.5%; in those two cases, the miss rate of about 2.3% likely dominates all other performance losses. In the next chapter, we will examine the relationship between the i7 CPI and cache misses, as well as other pipeline effects.

2.7 Fallacies and Pitfalls

As the most naturally quantitative of the computer architecture disciplines, memory hierarchy would seem to be less vulnerable to fallacies and pitfalls. Yet we were limited here not by lack of warnings, but by lack of space!

Fallacy *Predicting cache performance of one program from another.*

Figure 2.29 shows the instruction miss rates and data miss rates for three programs from the SPEC2000 benchmark suite as cache size varies. Depending on the

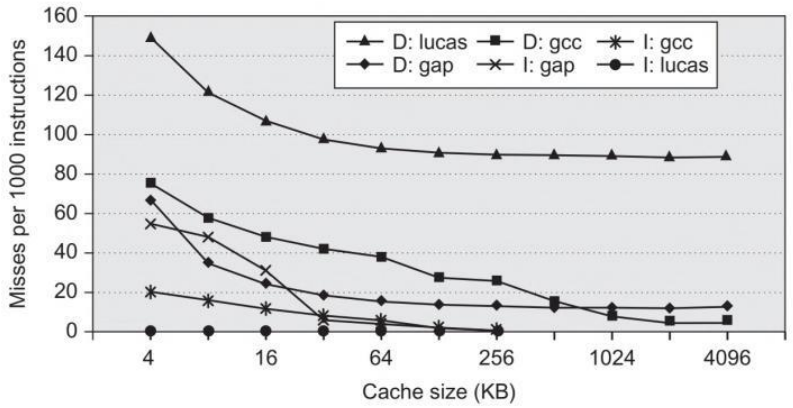


Figure 2.29 Instruction and data misses per 1000 instructions as cache size varies from 4 KiB to 4096 KiB. Instruction misses for gcc are 30,000–40,000 times larger than for lucas, and, conversely, data misses for lucas are 2–60 times larger than for gcc. The programs gap, gcc, and lucas are from the SPEC2000 benchmark suite.

program, the data misses per thousand instructions for a 4096 KiB cache are 9, 2, or 90, and the instruction misses per thousand instructions for a 4 KiB cache are 55, 19, or 0.0004. Commercial programs such as databases will have significant miss rates even in large second-level caches, which is generally not the case for the SPECCPU programs. Clearly, generalizing cache performance from one program to another is unwise. As Figure 2.24 reminds us, there is a great deal of variation, and even predictions about the relative miss rates of integer and floating-point-intensive programs can be wrong, as mcf and sphnix3 remind us!

Pitfall *Simulating enough instructions to get accurate performance measures of the memory hierarchy.*

There are really three pitfalls here. One is trying to predict performance of a large cache using a small trace. Another is that a program's locality behavior is not constant over the run of the entire program. The third is that a program's locality behavior may vary depending on the input.

Figure 2.30 shows the cumulative average instruction misses per thousand instructions for five inputs to a single SPEC2000 program. For these inputs, the average memory rate for the first 1.9 billion instructions is very different from the average miss rate for the rest of the execution.

Pitfall *Not delivering high memory bandwidth in a cache-based system.*

Caches help with average cache memory latency but may not deliver high memory bandwidth to an application that must go to main memory. The architect must design a high bandwidth memory behind the cache for such applications. We will revisit this pitfall in Chapters 4 and 5.

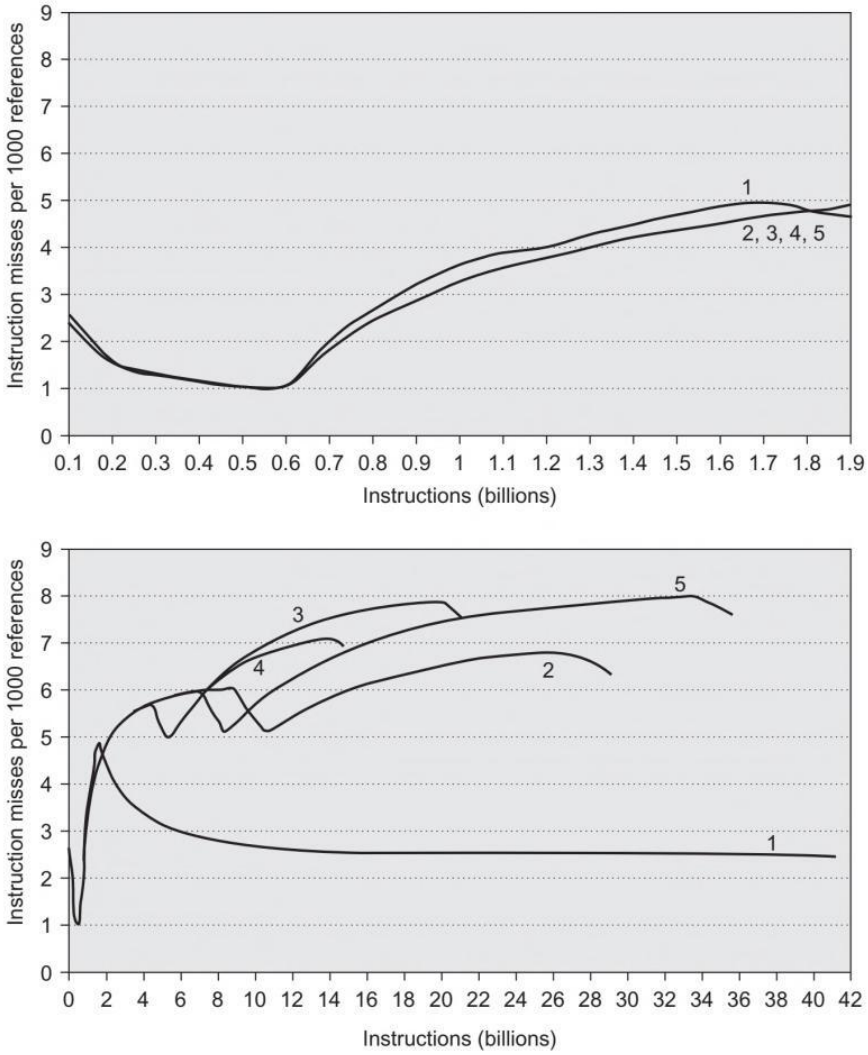


Figure 2.30 Instruction misses per 1000 references for five inputs to the perl benchmark in SPEC2000. There is little variation in misses and little difference between the five inputs for the first 1.9 billion instructions. Running to completion shows how misses vary over the life of the program and how they depend on the input. The top graph shows the running average misses for the first 1.9 billion instructions, which starts at about 2.5 and ends at about 4.7 misses per 1000 references for all five inputs. The bottom graph shows the running average misses to run to completion, which takes 16–41 billion instructions depending on the input. After the first 1.9 billion instructions, the misses per 1000 references vary from 2.4 to 7.9 depending on the input. The simulations were for the Alpha processor using separate L1 caches for instructions and data, each being two-way 64 KiB with LRU, and a unified 1 MiB direct-mapped L2 cache.

Pitfall *Implementing a virtual machine monitor on an instruction set architecture that wasn't designed to be virtualizable.*

Many architects in the 1970s and 1980s weren't careful to make sure that all instructions reading or writing information related to hardware resource information were privileged. This *laissez faire* attitude causes problems for VMMs for all of these architectures, including the 80x86, which we use here as an example.

Figure 2.31 describes the 18 instructions that cause problems for paravirtualization (Robin and Irvine, 2000). The two broad classes are instructions that

- read control registers in user mode that reveal that the guest operating system is running in a virtual machine (such as `POPF` mentioned earlier) and
- check protection as required by the segmented architecture but assume that the operating system is running at the highest privilege level.

Virtual memory is also challenging. Because the 80x86 TLBs do not support process ID tags, as do most RISC architectures, it is more expensive for the VMM and guest OSes to share the TLB; each address space change typically requires a TLB flush.

Problem category	Problem 80x86 instructions
Access sensitive registers without trapping when running in user mode	Store global descriptor table register (`SGDT`) Store local descriptor table register (`SLDT`) Store interrupt descriptor table register (`SIDT`) Store machine status word (`SMSW`) Push flags (`PUSHF`, `PUSHFD`) Pop flags (`POPF`, `POPFD`)
When accessing virtual memory mechanisms in user mode, instructions fail the 80x86 protection checks	Load access rights from segment descriptor (`LAR`) Load segment limit from segment descriptor (`LSL`) Verify if segment descriptor is readable (`VERR`) Verify if segment descriptor is writable (`VERW`) Pop to segment register (`POP CS`, `POP SS`, ...) Push segment register (`PUSH CS`, `PUSH SS`, ...) Far call to different privilege level (`CALL`) Far return to different privilege level (`RET`) Far jump to different privilege level (`JMP`) Software interrupt (`INT`) Store segment selector register (`STR`) Move to/from segment registers (`MOVE`)

Figure 2.31 Summary of 18 80x86 instructions that cause problems for virtualization (Robin and Irvine, 2000). The first five instructions of the top group allow a program in user mode to read a control register, such as a descriptor table register without causing a trap. The pop flags instruction modifies a control register with sensitive information but fails silently when in user mode. The protection checking of the segmented architecture of the 80x86 is the downfall of the bottom group because each of these instructions checks the privilege level implicitly as part of instruction execution when reading a control register. The checking assumes that the OS must be at the highest privilege level, which is not the case for guest VMs. Only the MOVE to segment register tries to modify control state, and protection checking foils it as well.

Virtualizing I/O is also a challenge for the 80x86, in part because it supports memory-mapped I/O and has separate I/O instructions, but more importantly because there are a very large number and variety of types of devices and device drivers of PCs for the VMM to handle. Third-party vendors supply their own drivers, and they may not properly virtualize. One solution for conventional VM implementations is to load real device drivers directly into the VMM.

To simplify implementations of VMMs on the 80x86, both AMD and Intel have proposed extensions to the architecture. Intel's VT-x provides a new execution mode for running VMs, a architected definition of the VM state, instructions to swap VMs rapidly, and a large set of parameters to select the circumstances where a VMM must be invoked. Altogether, VT-x adds 11 new instructions for the 80x86. AMD's Secure Virtual Machine (SVM) provides similar functionality.

After turning on the mode that enables VT-x support (via the new `VMXON` instruction), VT-x offers four privilege levels for the guest OS that are lower in priority than the original four (and fix issues like the problem with the `POPF` instruction mentioned earlier). VT-x captures all the states of a virtual machine in the Virtual Machine Control State (VMCS) and then provides atomic instructions to save and restore a VMCS. In addition to critical state, the VMCS includes configuration information to determine when to invoke the VMM and then specifically what caused the VMM to be invoked. To reduce the number of times the VMM must be invoked, this mode adds shadow versions of some sensitive registers and adds masks that check to see whether critical bits of a sensitive register will be changed before trapping. To reduce the cost of virtualizing virtual memory, AMD's SVM adds an additional level of indirection, called *nested page tables*, which makes shadow page tables unnecessary (see Section L.7 of Appendix L).

2.8 Concluding Remarks: Looking Ahead

Over the past thirty years there have been several predictions of the eminent [sic] cessation of the rate of improvement in computer performance. Every such prediction was wrong. They were wrong because they hinged on unstated assumptions that were overturned by subsequent events. So, for example, the failure to foresee the move from discrete components to integrated circuits led to a prediction that the speed of light would limit computer speeds to several orders of magnitude slower than they are now. Our prediction of the memory wall is probably wrong too but it suggests that we have to start thinking "out of the box."

Wm. A. Wulf and Sally A. McKee,
Hitting the Memory Wall: Implications of the Obvious,
Department of Computer Science, University of Virginia (December 1994).
This paper introduced the term *memory wall.*

The possibility of using a memory hierarchy dates back to the earliest days of general-purpose digital computers in the late 1940s and early 1950s. Virtual memory was introduced in research computers in the early 1960s and into IBM mainframes in the 1970s. Caches appeared around the same time. The basic concepts

have been expanded and enhanced over time to help close the access time gap between main memory and processors, but the basic concepts remain.

One trend that is causing a significant change in the design of memory hierarchies is a continued slowdown in both density and access time of DRAMs. In the past 15 years, both these trends have been observed and have been even more obvious over the past 5 years. While some increases in DRAM bandwidth have been achieved, decreases in access time have come much more slowly and almost vanished between DDR4 and DDR3. The end of Dennard scaling as well as a slowdown in Moore's Law both contributed to this situation. The trenched capacitor design used in DRAMs is also limiting its ability to scale. It may well be the case that packaging technologies such as stacked memory will be the dominant source of improvements in DRAM access bandwidth and latency.

Independently of improvements in DRAM, Flash memory has been playing a much larger role. In PMDs, Flash has dominated for 15 years and became the standard for laptops almost 10 years ago. In the past few years, many desktops have shipped with Flash as the primary secondary storage. Flash's potential advantage over DRAMs, specifically the absence of a per-bit transistor to control writing, is also its Achilles heel. Flash must use bulk erase-rewrite cycles that are considerably slower. As a result, although Flash has become the fastest growing form of secondary storage, SDRAMs still dominate for main memory.

Although phase-change materials as a basis for memory have been around for a while, they have never been serious competitors either for magnetic disks or for Flash. The recent announcement by Intel and Micron of the cross-point technology may change this. The technology appears to have several advantages over Flash, including the elimination of the slow erase-to-write cycle and greater longevity in terms. It could be that this technology will finally be the technology that replaces the electromechanical disks that have dominated bulk storage for more than 50 years!

For some years, a variety of predictions have been made about the coming memory wall (see previously cited quote and paper), which would lead to serious limits on processor performance. Fortunately, the extension of caches to multiple levels (from 2 to 4), more sophisticated refill and prefetch schemes, greater compiler and programmer awareness of the importance of locality, and tremendous improvements in DRAM bandwidth (a factor of over 150 times since the mid-1990s) have helped keep the memory wall at bay. In recent years, the combination of access time constraints on the size of L1 (which is limited by the clock cycle) and energy-related limitations on the size of L2 and L3 have raised new challenges. The evolution of the i7 processor class over 6–7 years illustrates this: the caches are the same size in the i7 6700 as they were in the first generation i7 processors! The more aggressive use of prefetching is an attempt to overcome the inability to increase L2 and L3. Off-chip L4 caches are likely to become more important because they are less energy-constrained than on-chip caches.

In addition to schemes relying on multilevel caches, the introduction of out-of-order pipelines with multiple outstanding misses has allowed available instruction-level parallelism to hide the memory latency remaining in a cache-based system. The introduction of multithreading and more thread-level parallelism takes this a step further by providing more parallelism and thus more latency-hiding

opportunities. It is likely that the use of instruction- and thread-level parallelism will be a more important tool in hiding whatever memory delays are encountered in modern multilevel cache systems.

One idea that periodically arises is the use of programmer-controlled scratchpad or other high-speed visible memories, which we will see are used in GPUs. Such ideas have never made the mainstream in general-purpose processors for several reasons: First, they break the memory model by introducing address spaces with different behavior. Second, unlike compiler-based or programmer-based cache optimizations (such as prefetching), memory transformations with scratchpads must completely handle the remapping from main memory address space to the scratchpad address space. This makes such transformations more difficult and limited in applicability. In GPUs (see Chapter 4), where local scratchpad memories are heavily used, the burden for managing them currently falls on the programmer. For domain-specific software systems that can use such memories, the performance gains are very significant. It is likely that HBM technologies will thus be used for caching in large, general-purpose computers and quite possibility as the main working memories in graphics and similar systems. As domain-specific architectures become more important in overcoming the limitations arising from the end of Dennard's Law and the slowdown in Moore's Law (see Chapter 7), scratchpad memories and vector-like register sets are likely to see more use.

The implications of the end of Dennard's Law affect both DRAM and processor technology. Thus, rather than a widening gulf between processors and main memory, we are likely to see a slowdown in both technologies, leading to slower overall growth rates in performance. New innovations in computer architecture and in related software that together increase performance and efficiency will be key to continuing the performance improvements seen over the past 50 years.

2.9 Historical Perspectives and References

In Section M.3 (available online) we examine the history of caches, virtual memory, and virtual machines. IBM plays a prominent role in the history of all three. References for further reading are included.

Case Studies and Exercises by Norman P. Jouppi, Rajeev Balasubramonian, Naveen Muralimanohar, and Sheng Li

Case Study 1: Optimizing Cache Performance via Advanced Techniques

Concepts illustrated by this case study

- Nonblocking Caches
- Compiler Optimizations for Caches
- Software and Hardware Prefetching
- Calculating Impact of Cache Performance on More Complex Processors

The transpose of a matrix interchanges its rows and columns; this concept is illustrated here:

$$\begin{bmatrix} A11 & A12 & A13 & A14 \\ A21 & A22 & A23 & A24 \\ A31 & A32 & A33 & A34 \\ A41 & A42 & A43 & A44 \end{bmatrix} \Rightarrow \begin{bmatrix} A11 & A21 & A31 & A41 \\ A12 & A22 & A32 & A42 \\ A13 & A23 & A33 & A43 \\ A14 & A24 & A34 & A44 \end{bmatrix}$$

Here is a simple C loop to show the transpose:

```
for (i = 0; i < 3; i++) {
 for (j = 0; j < 3; j++) {
 output[j][i] = input[i][j];
 }
}
```

Assume that both the input and output matrices are stored in the row major order (*row major order* means that the row index changes fastest). Assume that you are executing a 256·256 double-precision transpose on a processor with a 16 KB fully associative (don't worry about cache conflicts) least recently used (LRU) replacement L1 data cache with 64-byte blocks. Assume that the L1 cache misses or prefetches require 16 cycles and always hit in the L2 cache, and that the L2 cache can process a request every 2 processor cycles. Assume that each iteration of the preceding inner loop requires 4 cycles if the data are present in the L1 cache. Assume that the cache has a write-allocate fetch-on-write policy for write misses. Unrealistically, assume that writing back dirty cache blocks requires 0 cycles.

2.1 [10/15/15/12/20] <2.3> For the preceding simple implementation, this execution order would be nonideal for the input matrix; however, applying a loop interchange optimization would create a nonideal order for the output matrix. Because loop interchange is not sufficient to improve its performance, it must be blocked instead.

a. [10] <2.3> What should be the minimum size of the cache to take advantage of blocked execution?

b. [15] <2.3> How do the relative number of misses in the blocked and unblocked versions compare in the preceding minimum-sized cache?

c. [15] <2.3> Write code to perform a transpose with a block size parameter B that uses $B \cdot B$ blocks.

d. [12] <2.3> What is the minimum associativity required of the L1 cache for consistent performance independent of both arrays' position in memory?

e. [20] <2.3> Try out blocked and nonblocked 256·256 matrix transpositions on a computer. How closely do the results match your expectations based on what you know about the computer's memory system? Explain any discrepancies if possible.

2.2 [10] <2.3> Assume you are designing a hardware prefetcher for the preceding *unblocked* matrix transposition code. The simplest type of hardware prefetcher only prefetches sequential cache blocks after a miss. More complicated "nonunit stride" hardware prefetchers can analyze a miss reference stream and detect and prefetch nonunit strides. In contrast, software prefetching can determine nonunit strides as easily as it can determine unit strides. Assume prefetches write directly into the cache and that there is no "pollution" (overwriting data that must be used before the data that are prefetched). For best performance given a nonunit stride prefetcher, in the steady state of the inner loop, how many prefetches must be outstanding at a given time?

2.3 [15/20] <2.3> With software prefetching, it is important to be careful to have the prefetches occur in time for use but also to minimize the number of outstanding prefetches to live within the capabilities of the microarchitecture and minimize cache pollution. This is complicated by the fact that different processors have different capabilities and limitations.

a. [15] <2.3> Create a blocked version of the matrix transpose with software prefetching.

b. [20] <2.3> Estimate and compare the performance of the blocked and unblocked transpose codes both with and without software prefetching.

Case Study 2: Putting It All Together: Highly Parallel Memory Systems

Concept illustrated by this case study

■ Cross-Cutting Issues: The Design of Memory Hierarchies

The program in Figure 2.32 can be used to evaluate the behavior of a memory system. The key is having accurate timing and then having the program stride through memory to invoke different levels of the hierarchy. Figure 2.32 shows the code in C. The first part is a procedure that uses a standard utility to get an accurate measure of the user CPU time; this procedure may have to be changed to work on some systems. The second part is a nested loop to read and write memory at different strides and cache sizes. To get accurate cache timing, this code is repeated many times. The third part times the nested loop overhead only so that it can be subtracted from overall measured times to see how long the accesses were. The results are output in `.csv` file format to facilitate importing into spreadsheets. You may need to change `CACHE_MAX` depending on the question you are answering and the size of memory on the system you are measuring. Running the program in single-user mode or at least without other active applications will give more consistent results. The code in Figure 2.32 was derived from a program written by Andrea Dusseau at the University of California-Berkeley and was based on a detailed description found in Saavedra-Barrera (1992). It has been modified to fix a number of issues with more modern machines and to run under Microsoft

```
#include "stdafx.h"
#include <stdio.h>
#include <time.h>
#define ARRAY_MIN (1024) /* 1/4 smallest cache */
#define ARRAY_MAX (4096*4096) /* 1/4 largest cache */
int x[ARRAY_MAX]; /* array going to stride through */

double get_seconds() { /* routine to read time in seconds */
   __time64_t ltime;
   _time64( &ltime );
   return (double) ltime;
}
int label(int i) {/* generate text labels */
   if (i<1e3) printf("%1dB,",i);
   else if (i<1e6) printf("%1dK,",i/1024);
   else if (i<1e9) printf("%1dM,",i/1048576);
   else printf("%1dG,",i/1073741824);
   return 0;
}
int _tmain(int argc, _TCHAR* argv[]) {
int register nextstep, i, index, stride;
int csize;
double steps, tsteps;
double loadtime, lastsec, sec0, sec1, sec; /* timing variables */

/* Initialize output */
printf(" ,");
for (stride=1; stride <= ARRAY_MAX/2; stride=stride*2)
   label(stride*sizeof(int));
printf("\n");

/* Main loop for each configuration */
for (csize=ARRAY_MIN; csize <= ARRAY_MAX; csize=csize*2) {
   label(csize*sizeof(int)); /* print cache size this loop */
   for (stride=1; stride <= csize/2; stride=stride*2) {

      /* Lay out path of memory references in array */
      for (index=0; index < csize; index=index+stride)
         x[index] = index + stride; /* pointer to next */
      x[index-stride] = 0; /* loop back to beginning */

      /* Wait for timer to roll over */
      lastsec = get_seconds();
       sec0 = get_seconds(); while (sec0 == lastsec);

      /* Walk through path in array for twenty seconds */
      /* This gives 5% accuracy with second resolution */
      steps = 0.0; /* number of steps taken */
      nextstep = 0; /* start at beginning of path */
      sec0 = get_seconds(); /* start timer */
         { /* repeat until collect 20 seconds */
            (i=stride;i!=0;i=i-1) { /* keep samples same */
               nextstep = 0;
               do nextstep = x[nextstep]; /* dependency */
               while (nextstep != 0);
         }
         steps = steps + 1.0; /* count loop iterations */
         sec1 = get_seconds(); /* end timer */
      } while ((sec1 - sec0) < 20.0); /* collect 20 seconds */
      sec = sec1 - sec0;

      /* Repeat empty loop to loop subtract overhead */
      tsteps = 0.0; /* used to match no. while iterations */
      sec0 = get_seconds(); /* start timer */
         { /* repeat until same no. iterations as above */
            (i=stride;i!=0;i=i-1) { /* keep samples same */
               index = 0;
               do index = index + stride;
               while (index < csize);
         }
         tsteps = tsteps + 1.0;
         sec1 = get_seconds(); /* - overhead */
      } while (tsteps<steps); /* until = no. iterations */
      sec = sec - (sec1 - sec0);
      loadtime = (sec*1e9)/(steps*csize);
      /* write out results in .csv format for Excel */
      printf("%4.1f,", (loadtime<0.1) ? 0.1 : loadtime);
     }; /* end of inner for loop */
     printf("\n");
   }; /* end of outer for loop */
   return 0;
}
```

Figure 2.32 C program for evaluating memory system.

Visual C++. It can be downloaded from http://www.hpl.hp.com/research/cacti/aca_ch2_cs2.c.

The preceding program assumes that program addresses track physical addresses, which is true on the few machines that use virtually addressed caches, such as the Alpha 21264. In general, virtual addresses tend to follow physical addresses shortly after rebooting, so you may need to reboot the machine in order to get smooth lines in your results. To answer the following questions, assume that the sizes of all components of the memory hierarchy are powers of 2. Assume that the size of the page is much larger than the size of a block in a second-level cache (if there is one) and that the size of a second-level cache block is greater than or equal to the size of a block in a first-level cache. An example of the output of the program is plotted in Figure 2.33; the key lists the size of the array that is exercised.

2.4 [12/12/12/10/12] <2.6> Using the sample program results in Figure 2.33:

a. [12] <2.6> What are the overall size and block size of the second-level cache?

b. [12] <2.6> What is the miss penalty of the second-level cache?

c. [12] <2.6> What is the associativity of the second-level cache?

d. [10] <2.6> What is the size of the main memory?

e. [12] <2.6> What is the paging time if the page size is 4 KB?

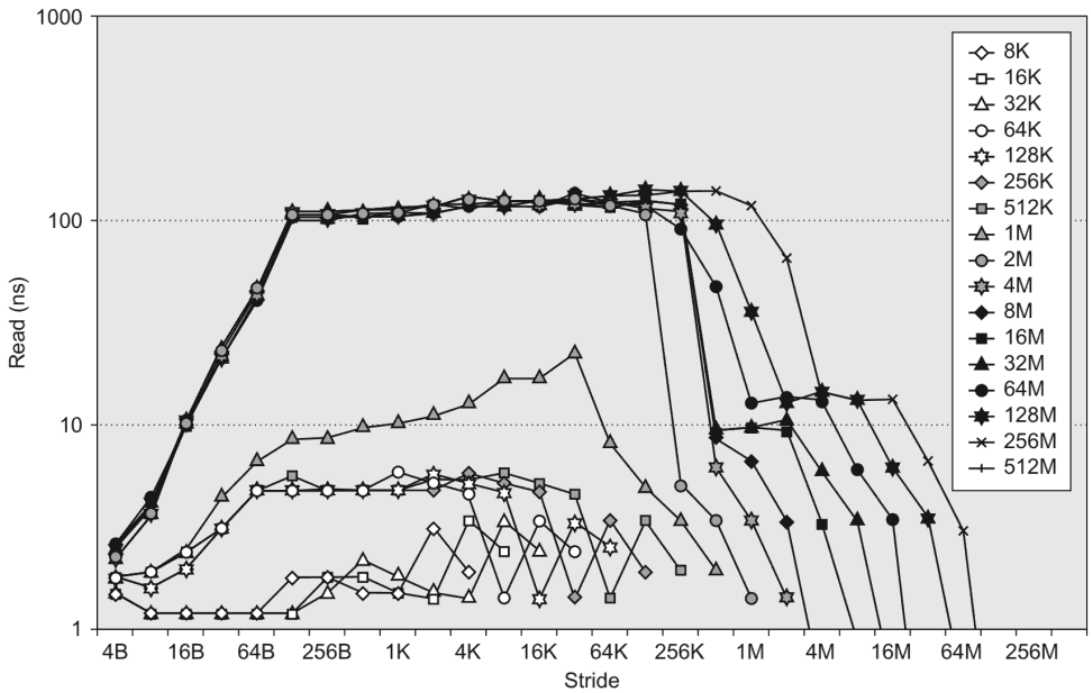


Figure 2.33 Sample results from program in Figure 2.32.

2.5 [12/15/15/20] <2.6> If necessary, modify the code in Figure 2.32 to measure the following system characteristics. Plot the experimental results with elapsed time on the *y*-axis and the memory stride on the *x*-axis. Use logarithmic scales for both axes, and draw a line for each cache size.

a. [12] <2.6> What is the system page size?

b. [15] <2.6> How many entries are there in the TLB?

c. [15] <2.6> What is the miss penalty for the TLB?

d. [20] <2.6> What is the associativity of the TLB?

2.6 [20/20] <2.6> In multiprocessor memory systems, lower levels of the memory hierarchy may not be able to be saturated by a single processor but should be able to be saturated by multiple processors working together. Modify the code in Figure 2.32, and run multiple copies at the same time. Can you determine:

a. [20] <2.6> How many actual processors are in your computer system and how many system processors are just additional multithreaded contexts?

b. [20] <2.6> How many memory controllers does your system have?

2.7 [20] <2.6> Can you think of a way to test some of the characteristics of an instruction cache using a program? *Hint*: The compiler may generate a large number of nonobvious instructions from a piece of code. Try to use simple arithmetic instructions of known length in your instruction set architecture (ISA).

Case Study 3: Studying the Impact of Various Memory System Organizations

Concepts illustrated by this case study

- DDR3 memory systems
- Impact of ranks, banks, row buffers on performance and power
- DRAM timing parameters

A processor chip typically supports a few DDR3 or DDR4 memory channels. We will focus on a single memory channel in this case study and explore how its performance and power are impacted by varying several parameters. Recall that the channel is populated with one or more DIMMs. Each DIMM supports one or more ranks—a rank is a collection of DRAM chips that work in unison to service a single command issued by the memory controller. For example, a rank may be composed of 16 DRAM chips, where each chip deals with a 4-bit input or output on every channel clock edge. Each such chip is referred to as a $\times 4$ (by four) chip. In other examples, a rank may be composed of 8×8 chips or 4×16 chips—note that in each case, a rank can handle data that are being placed on a 64-bit memory channel. A rank is itself partitioned into 8 (DDR3) or 16 (DDR4) banks. Each bank has a row buffer that essentially remembers the last row read out of a bank. Here's an example of a typical sequence of memory commands when performing a read from a bank:

(i) The memory controller issues a Precharge command to get the bank ready to access a new row. The precharge is completed after time tRP.

(ii) The memory controller then issues an Activate command to read the appropriate row out of the bank. The activation is completed after time tRCD and the row is deemed to be part of the row buffer.

(iii) The memory controller can then issue a column-read or CAS command that places a specific subset of the row buffer on the memory channel. After time CL, the first 64 bits of the data burst are placed on the memory channel. A burst typically includes eight 64-bit transfers on the memory channel, performed on the rising and falling edges of 4 memory clock cycles (referred to as transfer time).

(iv) If the memory controller wants to then access data in a different row of the bank, referred to as a row buffer miss, it repeats steps (i)–(iii). For now, we will assume that after CL has elapsed, the Precharge in step (i) can be issued; in some cases, an additional delay must be added, but we will ignore that delay here. If the memory controller wants to access another block of data in the same row, referred to as a row buffer hit, it simply issues another CAS command. Two back-to-back CAS commands have to be separated by at least 4 cycles so that the first data transfer is complete before the second data transfer can begin.

Note that a memory controller can issue commands to different banks in successive cycles so that it can perform many memory reads/writes in parallel and it is not sitting idle waiting for tRP, tRCD, and CL to elapse in a single bank. For the subsequent questions, assume that tRP = tRCD = CL = 13 ns, and that the memory channel frequency is 1 GHz, that is, a transfer time of 4 ns.

2.8 [10] <2.2> What is the read latency experienced by a memory controller on a row buffer miss?

2.9 [10] <2.2> What is the latency experienced by a memory controller on a row buffer hit?

2.10 [10] <2.2> If the memory channel supports only one bank and the memory access pattern is dominated by row buffer misses, what is the utilization of the memory channel?

2.11 [15] <2.2> Assuming a 100% row buffer miss rate, what is the minimum number of banks that the memory channel should support in order to achieve a 100% memory channel utilization?

2.12 [10] <2.2> Assuming a 50% row buffer miss rate, what is the minimum number of banks that the memory channel should support in order to achieve a 100% memory channel utilization?

2.13 [15] <2.2> Assume that we are executing an application with four threads and the threads exhibit zero spatial locality, that is, a 100% row buffer miss rate. Every 200 ns, each of the four threads simultaneously inserts a read operation into the

memory controller queue. What is the average memory latency experienced if the memory channel supports only one bank? What if the memory channel supported four banks?

2.14 [10] <2.2> From these questions, what have you learned about the benefits and downsides of growing the number of banks?

2.15 [20] <2.2> Now let's turn our attention to memory power. Download a copy of the Micron power calculator from this link: https://www.micron.com/~/media/documents/products/power-calculator/ddr3_power_calc.xlsm. This spreadsheet is preconfigured to estimate the power dissipation in a single 2 Gb ×8 DDR3 SDRAM memory chip manufactured by Micron. Click on the "Summary" tab to see the power breakdown in a single DRAM chip under default usage conditions (reads occupy the channel for 45% of all cycles, writes occupy the channel for 25% of all cycles, and the row buffer hit rate is 50%). This chip consumes 535 mW, and the breakdown shows that about half of that power is expended in Activate operations, about 38% in CAS operations, and 12% in background power. Next, click on the "System Config" tab. Modify the read/write traffic and the row buffer hit rate and observe how that changes the power profile. For example, what is the decrease in power when channel utilization is 35% (25% reads and 10% writes), or when row buffer hit rate is increased to 80%?

2.16 [20] <2.2> In the default configuration, a rank consists of eight ×8 2 Gb DRAM chips. A rank can also comprise16 × 4 chips or 4 × 16 chips. You can also vary the capacity of each DRAM chip—1 Gb, 2 Gb, and 4 Gb. These selections can be made in the "DDR3 Config" tab of the Micron power calculator. Tabulate the total power consumed for each rank organization. What is the most power-efficient approach to constructing a rank of a given capacity?

Exercises

2.17 [12/12/15] <2.3> The following questions investigate the impact of small and simple caches using CACTI and assume a 65 nm (0.065 m) technology. (CACTI is available in an online form at http://quid.hpl.hp.com:9081/cacti/.)

a. [12] <2.3> Compare the access times of 64 KB caches with 64-byte blocks and a single bank. What are the relative access times of two-way and four-way set associative caches compared to a direct mapped organization?

b. [12] <2.3> Compare the access times of four-way set associative caches with 64-byte blocks and a single bank. What are the relative access times of 32 and 64 KB caches compared to a 16 KB cache?

c. [15] <2.3> For a 64 KB cache, find the cache associativity between 1 and 8 with the lowest average memory access time given that misses per instruction for a certain workload suite is 0.00664 for direct-mapped, 0.00366 for two-way set associative, 0.000987 for four-way set associative, and 0.000266 for eight-way set associative cache. Overall, there are 0.3 data references per instruction. Assume cache misses take 10 ns in all models. To calculate the hit time in

cycles, assume the cycle time output using CACTI, which corresponds to the maximum frequency a cache can operate without any bubbles in the pipeline.

2.18 [12/15/15/10] <2.3> You are investigating the possible benefits of a way-predicting L1 cache. Assume that a 64 KB four-way set associative single-banked L1 data cache is the cycle time limiter in a system. For an alternative cache organization, you are considering a way-predicted cache modeled as a 64 KB direct-mapped cache with 80% prediction accuracy. Unless stated otherwise, assume that a mispredicted way access that hits in the cache takes one more cycle. Assume the miss rates and the miss penalties in question 2.8 part (c).

a. [12] <2.3> What is the average memory access time of the current cache (in cycles) versus the way-predicted cache?

b. [15] <2.3> If all other components could operate with the faster way-predicted cache cycle time (including the main memory), what would be the impact on performance from using the way-predicted cache?

c. [15] <2.3> Way-predicted caches have usually been used only for instruction caches that feed an instruction queue or buffer. Imagine that you want to try out way prediction on a data cache. Assume that you have 80% prediction accuracy and that subsequent operations (e.g., data cache access of other instructions, dependent operations) are issued assuming a correct way prediction. Thus a way misprediction necessitates a pipe flush and replay trap, which requires 15 cycles. Is the change in average memory access time per load instruction with data cache way prediction positive or negative, and how much is it?

d. [10] <2.3> As an alternative to way prediction, many large associative L2 caches serialize tag and data access so that only the required dataset array needs to be activated. This saves power but increases the access time. Use CACTI's detailed web interface for a 0.065 m process 1 MB four-way set associative cache with 64-byte blocks, 144 bits read out, 1 bank, only 1 read/write port, 30 bit tags, and ITRS-HP technology with global wires. What is the ratio of the access times for serializing tag and data access compared to parallel access?

2.19 [10/12] <2.3> You have been asked to investigate the relative performance of a banked versus pipelined L1 data cache for a new microprocessor. Assume a 64 KB two-way set associative cache with 64-byte blocks. The pipelined cache would consist of three pipe stages, similar in capacity to the Alpha 21264 data cache. A banked implementation would consist of two 32 KB two-way set associative banks. Use CACTI and assume a 65 nm (0.065 m) technology to answer the following questions. The cycle time output in the web version shows at what frequency a cache can operate without any bubbles in the pipeline.

a. [10] <2.3> What is the cycle time of the cache in comparison to its access time, and how many pipe stages will the cache take up (to two decimal places)?

b. [12] <2.3> Compare the area and total dynamic read energy per access of the pipelined design versus the banked design. State which takes up less area and which requires more power, and explain why that might be.

2.20 [12/15] <2.3> Consider the usage of critical word first and early restart on L2 cache misses. Assume a 1 MB L2 cache with 64-byte blocks and a refill path that is 16 bytes wide. Assume that the L2 can be written with 16 bytes every 4 processor cycles, the time to receive the first 16 byte block from the memory controller is 120 cycles, each additional 16 byte block from main memory requires 16 cycles, and data can be bypassed directly into the read port of the L2 cache. Ignore any cycles to transfer the miss request to the L2 cache and the requested data to the L1 cache.

a. [12] <2.3> How many cycles would it take to service an L2 cache miss with and without critical word first and early restart?

b. [15] <2.3> Do you think critical word first and early restart would be more important for L1 caches or L2 caches, and what factors would contribute to their relative importance?

2.21 [12/12] <2.3> You are designing a write buffer between a write-through L1 cache and a write-back L2 cache. The L2 cache write data bus is 16 B wide and can perform a write to an independent cache address every four processor cycles.

a. [12] <2.3> How many bytes wide should each write buffer entry be?

b. [15] <2.3> What speedup could be expected in the steady state by using a merging write buffer instead of a nonmerging buffer when zeroing memory by the execution of 64-bit stores if all other instructions could be issued in parallel with the stores and the blocks are present in the L2 cache?

c. [15] <2.3> What would the effect of possible L1 misses be on the number of required write buffer entries for systems with blocking and nonblocking caches?

2.22 [20] <2.1, 2.2, 2.3> A cache acts as a filter. For example, for every 1000 instructions of a program, an average of 20 memory accesses may exhibit low enough locality that they cannot be serviced by a 2 MB cache. The 2 MB cache is said to have an MPKI (misses per thousand instructions) of 20, and this will be largely true regardless of the smaller caches that precede the 2 MB cache. Assume the following cache/latency/MPKI values: 32 KB/1/100, 128 KB/2/80, 512 KB/4/50, 2 MB/8/40, 8 MB/16/10. Assume that accessing the off-chip memory system requires 200 cycles on average. For the following cache configurations, calculate the average time spent accessing the cache hierarchy. What do you observe about the downsides of a cache hierarchy that is too shallow or too deep?

a. 32 KB L1; 8 MB L2; off-chip memory

b. 32 KB L1; 512 KB L2; 8 MB L3; off-chip memory

c. 32 KB L1; 128 KB L2; 2 MB L3; 8 MB L4; off-chip memory

2.23 [15] <2.1, 2.2, 2.3> Consider a 16 MB 16-way L3 cache that is shared by two programs A and B. There is a mechanism in the cache that monitors cache miss rates for each program and allocates 1–15 ways to each program such that the overall number of cache misses is reduced. Assume that program A has an MPKI of 100 when it is assigned 1 MB of the cache. Each additional 1 MB assigned to program

A reduces the MPKI by 1. Program B has an MPKI of 50 when it is assigned 1 MB of cache; each additional 1 MB assigned to program B reduces its MPKI by 2. What is the best allocation of ways to programs A and B?

2.24 [20] <2.1, 2.6> You are designing a PMD and optimizing it for low energy. The core, including an 8 KB L1 data cache, consumes 1 W whenever it is not in hibernation. If the core has a perfect L1 cache hit rate, it achieves an average CPI of 1 for a given task, that is, 1000 cycles to execute 1000 instructions. Each additional cycle accessing the L2 and beyond adds a stall cycle for the core. Based on the following specifications, what is the size of L2 cache that achieves the lowest energy for the PMD (core, L1, L2, memory) for that given task?

a. The core frequency is 1 GHz, and the L1 has an MPKI of 100.

b. A 256 KB L2 has a latency of 10 cycles, an MPKI of 20, a background power of 0.2 W, and each L2 access consumes 0.5 nJ.

c. A 1 MB L2 has a latency of 20 cycles, an MPKI of 10, a background power of 0.8 W, and each L2 access consumes 0.7 nJ.

d. The memory system has an average latency of 100 cycles, a background power of 0.5 W, and each memory access consumes 35 nJ.

2.25 [15] <2.1, 2.6> You are designing a PMD that is optimized for low power. Qualitatively explain the impact on cache hierarchy (L2 and memory) power and overall application energy if you design an L2 cache with:

a. Small block size

b. Small cache size

c. High associativity

2.30 [10/10] <2.1, 2.2, 2.3> The ways of a set can be viewed as a priority list, ordered from high priority to low priority. Every time the set is touched, the list can be reorganized to change block priorities. With this view, cache management policies can be decomposed into three sub-policies: Insertion, Promotion, and Victim Selection. Insertion defines where newly fetched blocks are placed in the priority list. Promotion defines how a block's position in the list is changed every time it is touched (a cache hit). Victim Selection defines which entry of the list is evicted to make room for a new block when there is a cache miss.

a. Can you frame the LRU cache policy in terms of the Insertion, Promotion, and Victim Selection sub-policies?

b. Can you define other Insertion and Promotion policies that may be competitive and worth exploring further?

2.31 [15] <2.1, 2.3> In a processor that is running multiple programs, the last-level cache is typically shared by all the programs. This leads to interference, where one program's behavior and cache footprint can impact the cache available to other programs. First, this is a problem from a quality-of-service (QoS) perspective, where the interference leads to a program receiving fewer resources and lower

performance than promised, say by the operator of a cloud service. Second, this is a problem in terms of privacy. Based on the interference it sees, a program can infer the memory access patterns of other programs. This is referred to as a timing channel, a form of information leakage from one program to others that can be exploited to compromise data privacy or to reverse-engineer a competitor's algorithm. What policies can you add to your last-level cache so that the behavior of one program is immune to the behavior of other programs sharing the cache?

2.32 [15] <2.3> A large multimegabyte L3 cache can take tens of cycles to access because of the long wires that have to be traversed. For example, it may take 20 cycles to access a 16 MB L3 cache. Instead of organizing the 16 MB cache such that every access takes 20 cycles, we can organize the cache so that it is an array of smaller cache banks. Some of these banks may be closer to the processor core, while others may be further. This leads to nonuniform cache access (NUCA), where 2 MB of the cache may be accessible in 8 cycles, the next 2 MB in 10 cycles, and so on until the last 2 MB is accessed in 22 cycles. What new policies can you introduce to maximize performance in a NUCA cache?

2.33 [10/10/10] <2.2> Consider a desktop system with a processor connected to a 2 GB DRAM with *error-correcting code (ECC)*. Assume that there is only one memory channel of width 72 bits (64 bits for data and 8 bits for ECC).

a. [10] <2.2> How many DRAM chips are on the DIMM if 1 Gb DRAM chips are used, and how many data I/Os must each DRAM have if only one DRAM connects to each DIMM data pin?

b. [10] <2.2> What burst length is required to support 32 B L2 cache blocks?

c. [10] <2.2> Calculate the peak bandwidth for DDR2-667 and DDR2-533 DIMMs for reads from an active page excluding the ECC overhead.

2.34 [10/10] <2.2> A sample DDR2 SDRAM timing diagram is shown in Figure 2.34. tRCD is the time required to activate a row in a bank, and column address strobe (CAS) latency (CL) is the number of cycles required to read out a column in a row. Assume that the RAM is on a standard DDR2 DIMM with ECC, having 72 data lines. Also assume burst lengths of 8 that read out 8 bits, or a total of 64 B from the DIMM. Assume `tRCD = CAS (or CL) clock_frequency`, and `clock_frequency = transfers_per_second/2`. The on-chip latency

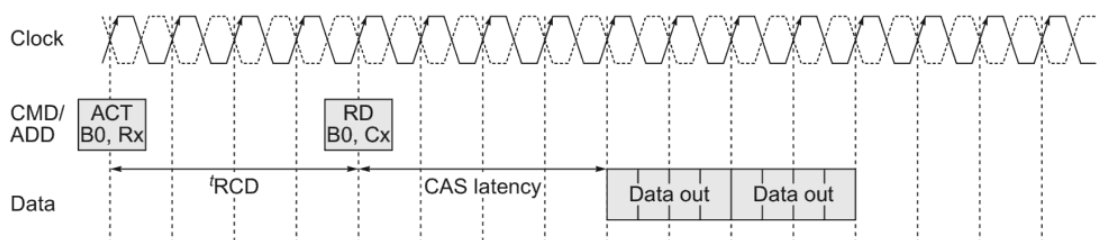


Figure 2.34 DDR2 SDRAM timing diagram.

on a cache miss through levels 1 and 2 and back, not including the DRAM access, is 20 ns.

a. [10] <2.2> How much time is required from presentation of the activate command until the last requested bit of data from the DRAM transitions from valid to invalid for the DDR2-667 1 Gb CL = 5 DIMM? Assume that for every request, we automatically prefetch another adjacent cache line in the same page.

b. [10] <2.2> What is the relative latency when using the DDR2-667 DIMM of a read requiring a bank activate versus one to an already open page, including the time required to process the miss inside the processor?

2.35 [15] <2.2> Assume that a DDR2-667 2 GB DIMM with CL = 5 is available for 130 and a DDR2-533 2 GB DIMM with CL = 4 is available for 100. Assume that two DIMMs are used in a system, and the rest of the system costs 800. Consider the performance of the system using the DDR2-667 and DDR2-533 DIMMs on a workload with 3.33 L2 misses per 1K instructions, and assume that 80% of all DRAM reads require an activate. What is the cost-performance of the entire system when using the different DIMMs, assuming only one L2 miss is outstanding at a time and an in-order core with a CPI of 1.5 not including L2 cache miss memory access time?

2.36 [12] <2.2> You are provisioning a server with eight-core 3 GHz CMP that can execute a workload with an overall CPI of 2.0 (assuming that L2 cache miss refills are not delayed). The L2 cache line size is 32 bytes. Assuming the system uses DDR2-667 DIMMs, how many independent memory channels should be provided so the system is not limited by memory bandwidth if the bandwidth required is sometimes twice the average? The workloads incur, on average, 6.67 L2 misses per 1 K instructions.

2.37 [15] <2.2> Consider a processor that has four memory channels. Should consecutive memory blocks be placed in the same bank, or should they be placed in different banks on different channels?

2.38 [12/12] <2.2> A large amount (more than a third) of DRAM power can be due to page activation (see http://download.micron.com/pdf/technotes/ddr2/TN4704.pdf and http://www.micron.com/systemcalc). Assume you are building a system with 2 GB of memory using either 8-bank 2 Gb ×8 DDR2 DRAMs or 8-bank 1 Gb ×8 DRAMs, both with the same speed grade. Both use a page size of 1 KB, and the last-level cache line size is 64 bytes. Assume that DRAMs that are not active are in precharged standby and dissipate negligible power. Assume that the time to transition from standby to active is not significant.

a. [12] <2.2> Which type of DRAM would be expected to provide the higher system performance? Explain why.

b. [12] <2.2> How does a 2 GB DIMM made of 1 Gb ×8 DDR2 DRAMs compare with a DIMM with similar capacity made of 1 Gb ×4 DDR2 DRAMs in terms of power?

2.39 [20/15/12] <2.2> To access data from a typical DRAM, we first have to activate the appropriate row. Assume that this brings an entire page of size 8 KB to the row buffer. Then we select a particular column from the row buffer. If subsequent accesses to DRAM are to the same page, then we can skip the activation step; otherwise, we have to close the current page and precharge the bitlines for the next activation. Another popular DRAM policy is to proactively close a page and precharge bitlines as soon as an access is over. Assume that every read or write to DRAM is of size 64 bytes and DDR bus latency (data from Figure 2.33) for sending 512 bits is Tddr.

a. [20] <2.2> Assuming DDR2-667, if it takes five cycles to precharge, five cycles to activate, and four cycles to read a column, for what value of the row buffer hit rate (r) will you choose one policy over another to get the best access time? Assume that every access to DRAM is separated by enough time to finish a random new access.

b. [15] <2.2> If 10% of the total accesses to DRAM happen back to back or contiguously without any time gap, how will your decision change?

c. [12] <2.2> Calculate the difference in average DRAM energy per access between the two policies using the previously calculated row buffer hit rate. Assume that precharging requires 2 nJ and activation requires 4 nJ and that 100 pJ/bit are required to read or write from the row buffer.

2.40 [15] <2.2> Whenever a computer is idle, we can either put it in standby (where DRAM is still active) or we can let it hibernate. Assume that, to hibernate, we have to copy just the contents of DRAM to a nonvolatile medium such as Flash. If reading or writing a cache line of size 64 bytes to Flash requires 2.56 J and DRAM requires 0.5 nJ, and if idle power consumption for DRAM is 1.6 W (for 8 GB), how long should a system be idle to benefit from hibernating? Assume a main memory of size 8 GB.

2.41 [10/10/10/10/10] <2.4> Virtual machines (VMs) have the potential for adding many beneficial capabilities to computer systems, such as improved total cost of ownership (TCO) or availability. Could VMs be used to provide the following capabilities? If so, how could they facilitate this?

a. [10] <2.4> Test applications in production environments using development machines?

b. [10] <2.4> Quick redeployment of applications in case of disaster or failure?

c. [10] <2.4> Higher performance in I/O-intensive applications?

d. [10] <2.4> Fault isolation between different applications, resulting in higher availability for services?

e. [10] <2.4> Performing software maintenance on systems while applications are running without significant interruption?

2.42 [10/10/12/12] <2.4> Virtual machines can lose performance from a number of events, such as the execution of privileged instructions, TLB misses, traps, and I/O.

Benchmark	Native	Pure	Para
Null call	0.04	0.96	0.50
Null I/O	0.27	6.32	2.91
Stat	1.10	10.69	4.14
Open/close	1.99	20.43	7.71
Install signal handler	0.33	7.34	2.89
Handle signal	1.69	19.26	2.36
Fork	56.00	513.00	164.00
Exec	316.00	2084.00	578.00
Fork + exec sh	1451.00	7790.00	2360.00

Figure 2.35 Early performance of various system calls under native execution, pure virtualization, and paravirtualization.

These events are usually handled in system code. Thus one way of estimating the slowdown when running under a VM is the percentage of application execution time in system versus user mode. For example, an application spending 10% of its execution in system mode might slow down by 60% when running on a VM. Figure 2.35 lists the early performance of various system calls under native execution, pure virtualization, and paravirtualization for LMbench using Xen on an Itanium system with times measured in microseconds (courtesy of Matthew Chapman of the University of New South Wales).

a. [10] <2.4> What types of programs would be expected to have smaller slowdowns when running under VMs?

b. [10] <2.4> If slowdowns were linear as a function of system time, given the preceding slowdown, how much slower would a program spending 20% of its execution in system time be expected to run?

c. [12] <2.4> What is the median slowdown of the system calls in the table above under pure virtualization and paravirtualization?

d. [12] <2.4> Which functions in the table above have the largest slowdowns? What do you think the cause of this could be?

2.43 [12] <2.4> Popek and Goldberg's definition of a virtual machine said that it would be indistinguishable from a real machine except for its performance. In this question, we will use that definition to find out if we have access to native execution on a processor or are running on a virtual machine. The Intel VT-x technology effectively provides a second set of privilege levels for the use of the virtual machine. What would a virtual machine running on top of another virtual machine have to do, assuming VT-x technology?

2.44 [20/25] <2.4> With the adoption of virtualization support on the x86 architecture, virtual machines are actively evolving and becoming mainstream. Compare and contrast the Intel VT-x and AMD's AMD-V virtualization technologies.

(Information on AMD-V can be found at http://sites.amd.com/us/business/it-solutions/virtualization/Pages/resources.aspx.)

a. [20] <2.4> Which one could provide higher performance for memory-intensive applications with large memory footprints?

b. [25] <2.4> Information on AMD's IOMMU support for virtualized I/O can be found at http://developer.amd.com/documentation/articles/pages/892006101.aspx. What do Virtualization Technology and an input/output memory management unit (IOMMU) do to improve virtualized I/O performance?

2.45 [30] <2.2, 2.3> Since instruction-level parallelism can also be effectively exploited on in-order superscalar processors and *very long instruction word (VLIW)* processors with speculation, one important reason for building an out-of-order (OOO) superscalar processor is the ability to tolerate unpredictable memory latency caused by cache misses. Thus you can think about hardware supporting OOO issue as being part of the memory system. Look at the floorplan of the Alpha 21264 in Figure 2.36 to find the relative area of the integer and floating-point issue queues and mappers versus the caches. The queues schedule instructions for issue,

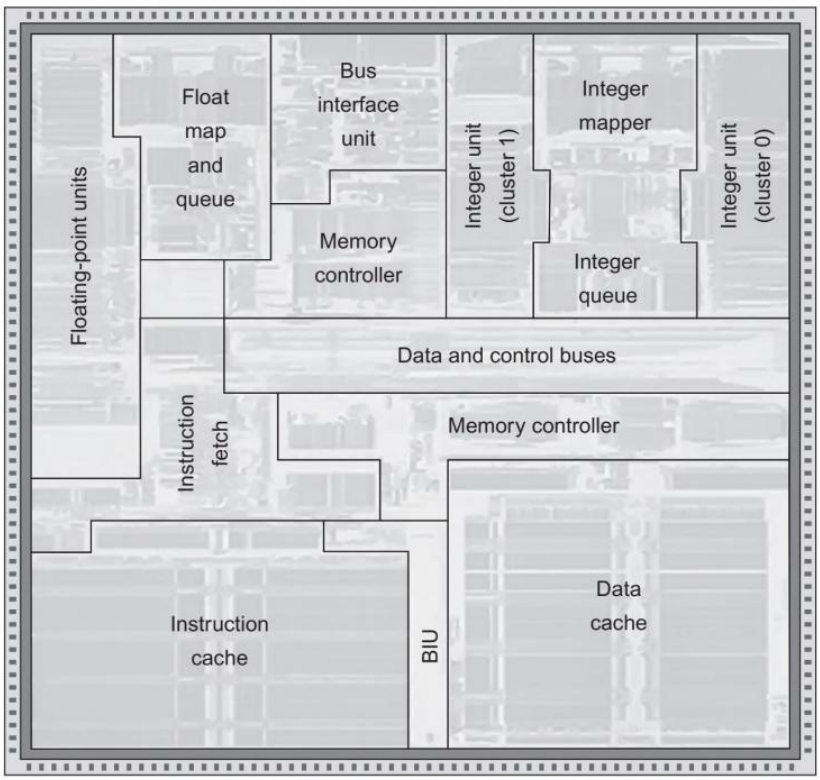


Figure 2.36 Floorplan of the Alpha 21264 [Kessler 1999].

and the mappers rename register specifiers. Therefore these are necessary additions to support OOO issue. The 21264 only has L1 data and instruction caches on chip, and they are both 64 KB two-way set associative. Use an OOO superscalar simulator such as SimpleScalar (http://www.cs.wisc.edu/~mscalar/simplescalar.html) on memory-intensive benchmarks to find out how much performance is lost if the area of the issue queues and mappers is used for additional L1 data cache area in an in-order superscalar processor, instead of OOO issue in a model of the 21264. Make sure the other aspects of the machine are as similar as possible to make the comparison fair. Ignore any increase in access or cycle time from larger caches and effects of the larger data cache on the floorplan of the chip. (Note that this comparison will not be totally fair, as the code will not have been scheduled for the in-order processor by the compiler.)

2.46 [15] <2.2, 2.7> As discussed in Section 2.7, the Intel i7 processor has an aggressive prefetcher. What are potential disadvantages in designing a prefetcher that is extremely aggressive?

2.47 [20/20/20] <2.6> The Intel performance analyzer VTune can be used to make many measurements of cache behavior. A free evaluation version of VTune on both Windows and Linux can be downloaded from http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/. The program (`aca_ch2_cs2.c`) used in Case Study 2 has been modified so that it can work with VTune out of the box on Microsoft Visual C++. The program can be downloaded from http://www.hpl.hp.com/research/cacti/aca_ch2_cs2_vtune.c. Special VTune functions have been inserted to exclude initialization and loop overhead during the performance analysis process. Detailed VTune setup directions are given in the README section in the program. The program keeps looping for 20 seconds for every configuration. In the following experiment, you can find the effects of data size on cache and overall processor performance. Run the program in VTune on an Intel processor with the input dataset sizes of 8 KB, 128 KB, 4 MB, and 32 MB, and keep a stride of 64 bytes (stride one cache line on Intel i7 processors). Collect statistics on overall performance and L1 data cache, L2, and L3 cache performance.

a. [20] <2.6> List the number of misses per 1K instruction of L1 data cache, L2, and L3 for each dataset size and your processor model and speed. Based on the results, what can you say about the L1 data cache, L2, and L3 cache sizes on your processor? Explain your observations.

b. [20] <2.6> List the *instructions per clock* (IPC) for each dataset size and your processor model and speed. Based on the results, what can you say about the L1, L2, and L3 miss penalties on your processor? Explain your observations.

c. [20] <2.6> Run the program in VTune with input dataset size of 8 KB and 128 KB on an Intel OOO processor. List the number of L1 data cache and L2 cache misses per 1K instructions and the CPI for both configurations. What can you say about the effectiveness of memory latency hiding techniques in high-performance OOO processors? *Hint*: You need to find the L1 data cache miss latency for your processor. For recent Intel i7 processors, it is approximately 11 cycles.

This page intentionally left blank

3

Instruction-Level Parallelism and Its Exploitation

"Who's first?"
"America."
"Who's second?"
"Sir, there is no second."

Dialog between two observers of the sailing race in 1851, later named "The America's Cup," which was the inspiration for John Cocke's naming of an IBM research processor as "America," the first superscalar processor, and a precursor to the PowerPC.

Thus, the IA-64 gambles that, in the future, power will not be the critical limitation, and massive resources…will not penalize clock speed, path length, or CPI factors. My view is clearly skeptical…

Marty Hopkins (2000), IBM Fellow and Early RISC pioneer commenting in 2000 on the new Intel Itanium, a joint development of Intel and HP. The Itanium used a static ILP approach (see Appendix H) and was a massive investment for Intel. It never accounted for more than 0.5% of Intel's microprocessor sales.

Computer Architecture. https://doi.org/10.1016/B978-0-12-811905-1.00003-1

3.1 Instruction-Level Parallelism: Concepts and Challenges

All processors since about 1985 have used pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called *instruction-level parallelism* (ILP), because the instructions can be evaluated in parallel. In this chapter and Appendix H, we look at a wide range of techniques for extending the basic pipelining concepts by increasing the amount of parallelism exploited among instructions.

This chapter is at a considerably more advanced level than the material on basic pipelining in Appendix C. If you are not thoroughly familiar with the ideas in Appendix C, you should review that appendix before venturing into this chapter.

We start this chapter by looking at the limitation imposed by data and control hazards and then turn to the topic of increasing the ability of the compiler and the processor to exploit parallelism. These sections introduce a large number of concepts, which we build on throughout this chapter and the next. While some of the more basic material in this chapter could be understood without all of the ideas in the first two sections, this basic material is important to later sections of this chapter.

There are two largely separable approaches to exploiting ILP: (1) an approach that relies on hardware to help discover and exploit the parallelism dynamically, and (2) an approach that relies on software technology to find parallelism statically at compile time. Processors using the dynamic, hardware-based approach, including all recent Intel and many ARM processors, dominate in the desktop and server markets. In the personal mobile device market, the same approaches are used in processors found in tablets and high-end cell phones. In the IOT space, where power and cost constraints dominate performance goals, designers exploit lower levels of instruction-level parallelism. Aggressive compiler-based approaches have been attempted numerous times beginning in the 1980s and most recently in the Intel Itanium series, introduced in 1999. Despite enormous efforts, such approaches have been successful only in domain-specific environments or in well-structured scientific applications with significant data-level parallelism.

In the past few years, many of the techniques developed for one approach have been exploited within a design relying primarily on the other. This chapter introduces the basic concepts and both approaches. A discussion of the limitations on ILP approaches is included in this chapter, and it was such limitations that directly led to the movement toward multicore. Understanding the limitations remains important in balancing the use of ILP and thread-level parallelism.

In this section, we discuss features of both programs and processors that limit the amount of parallelism that can be exploited among instructions, as well as the critical mapping between program structure and hardware structure, which is key to understanding whether a program property will actually limit performance and under what circumstances.

The value of the CPI (cycles per instruction) for a pipelined processor is the sum of the base CPI and all contributions from stalls:

$$\text{Pipeline CPI} = \text{Ideal pipeline CPI} + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls}$$

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	C.2
Simple branch scheduling and prediction	Control hazard stalls	C.2
Basic compiler pipeline scheduling	Data hazard stalls	C.2, 3.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	C.7
Loop unrolling	Control hazard stalls	3.2
Advanced branch prediction	Control stalls	3.3
Dynamic scheduling with renaming	Stalls from data hazards, output dependences, and antidependences	3.4
Hardware speculation	Data hazard and control hazard stalls	3.6
Dynamic memory disambiguation	Data hazard stalls with memory	3.6
Issuing multiple instructions per cycle	Ideal CPI	3.7, 3.8
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls	H.2, H.3
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls	H.4, H.5

Figure 3.1 The major techniques examined in Appendix C, Chapter 3, and Appendix H are shown together with the component of the CPI equation that the technique affects.

The *ideal pipeline CPI* is a measure of the maximum performance attainable by the implementation. By reducing each of the terms of the right-hand side, we decrease the overall pipeline CPI or, alternatively, increase the IPC (instructions per clock). The preceding equation allows us to characterize various techniques by what component of the overall CPI a technique reduces. Figure 3.1 shows the techniques we examine in this chapter and in Appendix H, as well as the topics covered in the introductory material in Appendix C. In this chapter, we will see that the techniques we introduce to decrease the ideal pipeline CPI can increase the importance of dealing with hazards.

What Is Instruction-Level Parallelism?

All the techniques in this chapter exploit parallelism among instructions. The amount of parallelism available within a *basic block*—a straight-line code sequence with no branches in except to the entry and no branches out except at the exit—is quite small. For typical RISC programs, the average dynamic branch frequency is often between 15% and 25%, meaning that between three and six instructions execute between a pair of branches. Because these instructions are likely to depend upon one another, the amount of overlap we can exploit within a basic block is likely to be less than the average basic block size. To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks.

The simplest and most common way to increase the ILP is to exploit parallelism among iterations of a loop. This type of parallelism is often called *loop-level*

parallelism. Here is a simple example of a loop that adds two 1000-element arrays and is completely parallel:

```
for (i=0; i<=999; i=i+1)
        x[i] = x[i] + y[i];
```

Every iteration of the loop can overlap with any other iteration, although within each loop iteration, there is little or no opportunity for overlap.

We will examine a number of techniques for converting such loop-level parallelism into instruction-level parallelism. Basically, such techniques work by unrolling the loop either statically by the compiler (as in the next section) or dynamically by the hardware (as in Sections 3.5 and 3.6).

An important alternative method for exploiting loop-level parallelism is the use of SIMD in both vector processors and graphics processing units (GPUs), both of which are covered in Chapter 4. A SIMD instruction exploits data-level parallelism by operating on a small to moderate number of data items in parallel (typically two to eight). A vector instruction exploits data-level parallelism by operating on many data items in parallel using both parallel execution units and a deep pipeline. For example, the preceding code sequence, which in simple form requires seven instructions per iteration (two loads, an add, a store, two address updates, and a branch) for a total of 7000 instructions, might execute in one-quarter as many instructions in some SIMD architecture where four data items are processed per instruction. On some vector processors, this sequence might take only four instructions: two instructions to load the vectors `x` and `y` from memory, one instruction to add the two vectors, and an instruction to store back the result vector. Of course, these instructions would be pipelined and have relatively long latencies, but these latencies may be overlapped.

Data Dependences and Hazards

Determining how one instruction depends on another is critical to determining how much parallelism exists in a program and how that parallelism can be exploited. In particular, to exploit instruction-level parallelism, we must determine which instructions can be executed in parallel. If two instructions are *parallel*, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls, assuming the pipeline has sufficient resources (and thus no structural hazards exist). If two instructions are dependent, they are not parallel and must be executed in order, although they may often be partially overlapped. The key in both cases is to determine whether an instruction is dependent on another instruction.

Data Dependences

There are three different types of dependences: *data dependences* (also called true data dependences), *name dependences*, and *control dependences*. An instruction *j* is *data-dependent* on instruction *i* if either of the following holds:

- Instruction *i* produces a result that may be used by instruction *j*.
- Instruction *j* is data-dependent on instruction *k*, and instruction *k* is data-dependent on instruction *i*.

The second condition simply states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions. This dependence chain can be as long as the entire program. Note that a dependence within a single instruction (such as `add x1,x1,x1`) is not considered a dependence.

For example, consider the following RISC-V code sequence that increments a vector of values in memory (starting at `0(x1)` ending with the last element at `0(x2)`) by a scalar in register `f2`.

```
Loop:  fld     f0,0(x1)    //f0=array element
       fadd.d  f4,f0,f2    //add scalar in f2
       fsd     f4,0(x1)    //store result
       addi    x1,x1,−8    //decrement pointer 8 bytes
       bne     x1,x2,Loop  //branch x1≠x2
```

The data dependences in this code sequence involve both floating-point data:

```
Loop: fld     f0,0(x1)   //f0=array element
      fadd.d  f4,f0,f2   //add scalar in f2
      fsd     f4,0(x1)   //store result
```

and integer data:

```
addi     x1,x1,-8  //decrement pointer
                   //8 bytes (per DW)
bne      x1,x2,Loop//branch x1ax2
```

In both of the preceding dependent sequences, as shown by the arrows, each instruction depends on the previous one. The arrows here and in following examples show the order that must be preserved for correct execution. The arrow points from an instruction that must precede the instruction that the arrowhead points to.

If two instructions are data-dependent, they must execute in order and cannot execute simultaneously or be completely overlapped. The dependence implies that there would be a chain of one or more data hazards between the two instructions. (See Appendix C for a brief description of data hazards, which we will define precisely in a few pages.) Executing the instructions simultaneously will cause a processor with pipeline interlocks (and a pipeline depth longer than the distance between the instructions in cycles) to detect a hazard and stall, thereby reducing or eliminating the overlap. In a processor without interlocks that relies on compiler scheduling, the compiler cannot schedule dependent instructions in such a way that

they completely overlap because the program will not execute correctly. The presence of a data dependence in an instruction sequence reflects a data dependence in the source code from which the instruction sequence was generated. The effect of the original data dependence must be preserved.

Dependences are a property of *programs.* Whether a given dependence results in an actual hazard being detected and whether that hazard actually causes a stall are properties of the *pipeline organization.* This difference is critical to understanding how instruction-level parallelism can be exploited.

A data dependence conveys three things: (1) the possibility of a hazard, (2) the order in which results must be calculated, and (3) an upper bound on how much parallelism can possibly be exploited. Such limits are explored in a pitfall on page 262 and in Appendix H in more detail.

Because a data dependence can limit the amount of instruction-level parallelism we can exploit, a major focus of this chapter is overcoming these limitations. A dependence can be overcome in two different ways: (1) maintaining the dependence but avoiding a hazard, and (2) eliminating a dependence by transforming the code. Scheduling the code is the primary method used to avoid a hazard without altering a dependence, and such scheduling can be done both by the compiler and by the hardware.

A data value may flow between instructions either through registers or through memory locations. When the data flow occurs through a register, detecting the dependence is straightforward because the register names are fixed in the instructions, although it gets more complicated when branches intervene and correctness concerns force a compiler or hardware to be conservative.

Dependences that flow through memory locations are more difficult to detect because two addresses may refer to the same location but look different: For example, `100(x4)` and `20(x6)` may be identical memory addresses. In addition, the effective address of a load or store may change from one execution of the instruction to another (so that `20(x4)` and `20(x4)` may be different), further complicating the detection of a dependence.

In this chapter, we examine hardware for detecting data dependences that involve memory locations, but we will see that these techniques also have limitations. The compiler techniques for detecting such dependences are critical in uncovering loop-level parallelism.

Name Dependences

The second type of dependence is a *name dependence.* A name dependence occurs when two instructions use the same register or memory location, called a *name*, but there is no flow of data between the instructions associated with that name. There are two types of name dependences between an instruction *i* that *precedes* instruction *j* in program order:

1. An *antidependence* between instruction *i* and instruction *j* occurs when instruction *j* writes a register or memory location that instruction *i* reads. The original

ordering must be preserved to ensure that *i* reads the correct value. In the example on page 171, there is an antidependence between `fsd` and `addi` on register `x1`.

2. An *output dependence* occurs when instruction *i* and instruction *j* write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction *j*.

Both antidependences and output dependences are name dependences, as opposed to true data dependences, because there is no value being transmitted between the instructions. Because a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict.

This renaming can be more easily done for register operands, where it is called *register renaming*. Register renaming can be done either statically by a compiler or dynamically by the hardware. Before describing dependences arising from branches, let's examine the relationship between dependences and pipeline data hazards.

Data Hazards

A hazard exists whenever there is a name or data dependence between instructions, and they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence. Because of the dependence, we must preserve what is called *program order*—that is, the order that the instructions would execute in if executed sequentially one at a time as determined by the original source program. The goal of both our software and hardware techniques is to exploit parallelism by preserving program order *only where it affects the outcome of the program.* Detecting and avoiding hazards ensures that necessary program order is preserved.

Data hazards, which are informally described in Appendix C, may be classified as one of three types, depending on the order of read and write accesses in the instructions. By convention, the hazards are named by the ordering in the program that must be preserved by the pipeline. Consider two instructions *i* and *j,* with *i* preceding *j* in program order. The possible data hazards are

- RAW *(read after write)—j* tries to read a source before *i* writes it, so *j* incorrectly gets the *old* value. This hazard is the most common type and corresponds to a true data dependence. Program order must be preserved to ensure that *j* receives the value from *i*.
- WAW *(write after write)—j* tries to write an operand before it is written by *i*. The writes end up being performed in the wrong order, leaving the value written by *i* rather than the value written by *j* in the destination. This hazard corresponds to an output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

- WAR *(write after read)—j* tries to write a destination before it is read by *i,* so *i* incorrectly gets the *new* value. This hazard arises from an antidependence (or name dependence). WAR hazards cannot occur in most static issue pipelines—even deeper pipelines or floating-point pipelines—because all reads are early (in ID in the pipeline in Appendix C) and all writes are late (in WB in the pipeline in Appendix C). A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline *and* other instructions that read a source late in the pipeline, or when instructions are reordered, as we will see in this chapter.

Note that the RAR *(read after read)* case is not a hazard.

Control Dependences

The last type of dependence is a *control dependence.* A control dependence determines the ordering of an instruction, *i,* with respect to a branch instruction so that instruction *i* is executed in correct program order and only when it should be. Every instruction, except for those in the first basic block of the program, is control-dependent on some set of branches, and in general, these control dependences must be preserved to preserve program order. One of the simplest examples of a control dependence is the dependence of the statements in the "then" part of an if statement on the branch. For example, in the code segment

```
if p1 {
      S1;
};
if p2 {
      S2;
}
```

`S1` is control-dependent on `p1`, and `S2` is control-dependent on `p2` but not on `p1`.

In general, two constraints are imposed by control dependences:

1. An instruction that is control-dependent on a branch cannot be moved *before* the branch so that its execution *is no longer controlled* by the branch. For example, we cannot take an instruction from the then portion of an if statement and move it before the if statement.
2. An instruction that is not control-dependent on a branch cannot be moved *after* the branch so that its execution *is controlled* by the branch. For example, we cannot take a statement before the if statement and move it into the then portion.

When processors preserve strict program order, they ensure that control dependences are also preserved. We may be willing to execute instructions that should not have been executed, however, thereby violating the control dependences, *if* we

can do so without affecting the correctness of the program. Thus control dependence is not the critical property that must be preserved. Instead, the two properties critical to program correctness—and normally preserved by maintaining both data and control dependences—are the *exception behavior* and the *data flow*.

Preserving the exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program. Often this is relaxed to mean that the reordering of instruction execution must not cause any new exceptions in the program. A simple example shows how maintaining the control and data dependences can prevent such situations. Consider this code sequence:

```
        add  x2,x3,x4
        beq  x2,x0,L1
        ld   x1,0(x2)
L1:
```

In this case, it is easy to see that if we do not maintain the data dependence involving `x2`, we can change the result of the program. Less obvious is the fact that if we ignore the control dependence and move the load instruction before the branch, the load instruction may cause a memory protection exception. Notice that *no data dependence* prevents us from interchanging the `beqz` and the `ld`; it is only the control dependence. To allow us to reorder these instructions (and still preserve the data dependence), we want to just ignore the exception when the branch is taken. In Section 3.6, we will look at a hardware technique, *speculation*, which allows us to overcome this exception problem. Appendix H looks at software techniques for supporting speculation.

The second property preserved by maintenance of data dependences and control dependences is the data flow. The *data flow* is the actual flow of data values among instructions that produce results and those that consume them. Branches make the data flow dynamic because they allow the source of data for a given instruction to come from many points. Put another way, it is insufficient to just maintain data dependences because an instruction may be data-dependent on more than one predecessor. Program order is what determines which predecessor will actually deliver a data value to an instruction. Program order is ensured by maintaining the control dependences.

For example, consider the following code fragment:

```
        add  x1,x2,x3
        beq  x4,x0,L
        sub  x1,x5,x6
L:      ...
        or   x7,x1,x8
```

In this example, the value of `x1` used by the `or` instruction depends on whether the branch is taken or not. Data dependence alone is not sufficient to preserve correctness. The `or` instruction is data-dependent on both the `add` and `sub` instructions, but preserving that order alone is insufficient for correct execution.

Instead, when the instructions execute, the data flow must be preserved: If the branch is not taken, then the value of `x1` computed by the `sub` should be used by the `or`, and if the branch is taken, the value of `x1` computed by the `add` should be used by the `or`. By preserving the control dependence of the `or` on the branch, we prevent an illegal change to the data flow. For similar reasons, the `sub` instruction cannot be moved above the branch. Speculation, which helps with the exception problem, will also allow us to lessen the impact of the control dependence while still maintaining the data flow, as we will see in Section 3.6.

Sometimes we can determine that violating the control dependence cannot affect either the exception behavior or the data flow. Consider the following code sequence:

```
        add  x1,x2,x3
        beq  x12,x0,skip
        sub  x4,x5,x6
        add  x5,x4,x9
skip:   or   x7,x8,x9
```

Suppose we knew that the register destination of the `sub` instruction (`x4`) was unused after the instruction labeled `skip`. (The property of whether a value will be used by an upcoming instruction is called *liveness*.) If `x4` were unused, then changing the value of `x4` just before the branch would not affect the data flow because `x4` would be *dead* (rather than live) in the code region after `skip`. Thus, if `x4` were dead and the existing `sub` instruction could not generate an exception (other than those from which the processor resumes the same process), we could move the `sub` instruction before the branch because the data flow could not be affected by this change.

If the branch is taken, the `sub` instruction will execute and will be useless, but it will not affect the program results. This type of code scheduling is also a form of speculation, often called software speculation, because the compiler is betting on the branch outcome; in this case, the bet is that the branch is usually not taken. More ambitious compiler speculation mechanisms are discussed in Appendix H. Normally, it will be clear when we say speculation or speculative whether the mechanism is a hardware or software mechanism; when it is not clear, it is best to say "hardware speculation" or "software speculation."

Control dependence is preserved by implementing control hazard detection that causes control stalls. Control stalls can be eliminated or reduced by a variety of hardware and software techniques, which we examine in Section 3.3.

3.2 Basic Compiler Techniques for Exposing ILP

This section examines the use of simple compiler technology to enhance a processor's ability to exploit ILP. These techniques are crucial for processors that use static issue or static scheduling. Armed with this compiler technology, we will shortly examine the design and performance of processors using static issuing. Appendix H will investigate more sophisticated compiler and associated hardware schemes designed to enable a processor to exploit more instruction-level parallelism.

Basic Pipeline Scheduling and Loop Unrolling

To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline. To avoid a pipeline stall, the execution of a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction. A compiler's ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline. Figure 3.2 shows the FP unit latencies we assume in this chapter, unless different latencies are explicitly stated. We assume the standard five-stage integer pipeline so that branches have a delay of one clock cycle. We assume that the functional units are fully pipelined or replicated (as many times as the pipeline depth) so that an operation of any type can be issued on every clock cycle and there are no structural hazards.

In this section, we look at how the compiler can increase the amount of available ILP by transforming loops. This example serves both to illustrate an important technique as well as to motivate the more powerful program transformations described in Appendix H. We will rely on the following code segment, which adds a scalar to a vector:

```
for (i=999; i>=0; i=i–1)
        x[i] = x[i] + s;
```

We can see that this loop is parallel by noticing that the body of each iteration is independent. We formalize this notion in Appendix H and describe how we can test whether loop iterations are independent at compile time. First, let's look at the performance of this loop, which shows how we can use the parallelism to improve its performance for a RISC-V pipeline with the preceding latencies.

The first step is to translate the preceding segment to RISC-V assembly language. In the following code segment, `x1` is initially the address of the element in the array with the highest address, and `f2` contains the scalar value *s*. Register `x2` is precomputed so that `Regs[x2]+8` is the address of the last element to operate on.

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Figure 3.2 Latencies of FP operations used in this chapter. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is 0 because the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0 (which includes ALU operation to branch).

The straightforward RISC-V code, not scheduled for the pipeline, looks like this:

```
Loop:  fld     f0,0(x1)      //f0=array element
       fadd.d  f4,f0,f2      //add scalar in f2
       fsd     f4,0(x1)      //store result
       addi    x1,x1,−8      //decrement pointer
                             //8 bytes (per DW)
       bne     x1,x2,Loop    //branch x1≠x2
```

Let's start by seeing how well this loop will run when it is scheduled on a simple pipeline for RISC-V with the latencies in Figure 3.2.

Example Show how the loop would look on RISC-V, both scheduled and unscheduled, including any stalls or idle clock cycles. Schedule for delays from floating-point operations.

Answer Without any scheduling, the loop will execute as follows, taking nine cycles:

```
                                    Clock cycle issued
Loop:  fld     f0,0(x1)                    1
       stall                               2
       fadd.d  f4,f0,f2                    3
       stall                               4
       stall                               5
       fsd     f4,0(x1)                    6
       addi    x1,x1,−8                    7
       bne     x1,x2,Loop                  8
```

We can schedule the loop to obtain only two stalls and reduce the time to seven cycles:

```
Loop:  fld     f0,0(x1)
       addi    x1,x1,−8
       fadd.d  f4,f0,f2
       stall
       stall
       fsd     f4,8(x1)
       bne     x1,x2,Loop
```

The stalls after `fadd.d` are for use by the `fsd`, and repositioning the `addi` prevents the stall after the `fld`.

In the previous example, we complete one loop iteration and store back one array element every seven clock cycles, but the actual work of operating on the array element takes just three (the load, add, and store) of those seven clock cycles.

The remaining four clock cycles consist of loop overhead—the `addi` and `bne`—and two stalls. To eliminate these four clock cycles, we need to get more operations relative to the number of overhead instructions.

A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is *loop unrolling*. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.

Loop unrolling can also be used to improve scheduling. Because it eliminates the branch, it allows instructions from different iterations to be scheduled together. In this case, we can eliminate the data use stalls by creating additional independent instructions within the loop body. If we simply replicated the instructions when we unrolled the loop, the resulting use of the same registers could prevent us from effectively scheduling the loop. Thus we will want to use different registers for each iteration, increasing the required number of registers.

Example Show our loop unrolled so that there are four copies of the loop body, assuming `x1 – x2` (that is, the size of the array) is initially a multiple of 32, which means that the number of loop iterations is a multiple of 4. Eliminate any obviously redundant computations and do not reuse any of the registers.

Answer Here is the result after merging the `addi` instructions and dropping the unnecessary `bne` operations that are duplicated during unrolling. Note that `x2` must now be set so that `Regs[x2]+32` is the starting address of the last four elements.

```
Loop:  fld     f0,0(x1)
       fadd.d  f4,f0,f2
       fsd     f4,0(x1)      //drop addi & bne
       fld     f6,-8(x1)
       fadd.d  f8,f6,f2
       fsd     f8,-8(x1)     //drop addi & bne
       fld     f0,-16(x1)
       fadd.d  f12,f0,f2
       fsd     f12,-16(x1)   //drop addi & bne
       fld     f14,-24(x1)
       fadd.d  f16,f14,f2
       fsd     f16,-24(x1)
       addi    x1,x1,-32
       bne     x1,x2,Loop
```

We have eliminated three branches and three decrements of `x1`. The addresses on the loads and stores have been compensated to allow the `addi` instructions on `x1` to be merged. This optimization may seem trivial, but it is not; it requires symbolic substitution and simplification. Symbolic substitution and simplification will rearrange expressions so as to allow constants to be collapsed, allowing an expression such as $((i+1)+1)$ to be rewritten as $(i+(1+1))$ and then simplified to $(i+2)$.

We will see more general forms of these optimizations that eliminate dependent computations in Appendix H.

Without scheduling, every FP load or operation in the unrolled loop is followed by a dependent operation and thus will cause a stall. This unrolled loop will run in 26 clock cycles—each `fld` has 1 stall, each `fadd.d` has 2, plus 14 instruction issue cycles—or 6.5 clock cycles for each of the four elements, but it can be scheduled to improve performance significantly. Loop unrolling is normally done early in the compilation process so that redundant computations can be exposed and eliminated by the optimizer.

In real programs, we do not usually know the upper bound on the loop. Suppose it is *n,* and we want to unroll the loop to make *k* copies of the body. Instead of a single unrolled loop, we generate a pair of consecutive loops. The first executes (*n* mod *k*) times and has a body that is the original loop. The second is the unrolled body surrounded by an outer loop that iterates (*n*/*k*) times. (As we will see in Chapter 4, this technique is similar to a technique called *strip mining*, used in compilers for vector processors.) For large values of *n*, most of the execution time will be spent in the unrolled loop body.

In the previous example, unrolling improves the performance of this loop by eliminating overhead instructions, although it increases code size substantially. How will the unrolled loop perform when it is scheduled for the pipeline described earlier?

Example Show the unrolled loop in the previous example after it has been scheduled for the pipeline with the latencies in Figure 3.2.

Answer

```
Loop:  fld     f0,0(x1)
       fld     f6,-8(x1)
       fld     f0,-16(x1)
       fld     f14,-24(x1)
       fadd.d  f4,f0,f2
       fadd.d  f8,f6,f2
       fadd.d  f12,f0,f2
       fadd.d  f16,f14,f2
       fsd     f4,0(x1)
       fsd     f8,-8(x1)
       fsd     f12,16(x1)
       fsd     f16,8(x1)
       addi    x1,x1,-32
       bne     x1,x2,Loop
```

The execution time of the unrolled loop has dropped to a total of 14 clock cycles, or 3.5 clock cycles per element, compared with 8 cycles per element before any unrolling or scheduling and 6.5 cycles when unrolled but not scheduled.

The gain from scheduling on the unrolled loop is even larger than on the original loop. This increase arises because unrolling the loop exposes more computation that can be scheduled to minimize the stalls; the preceding code has no stalls. Scheduling the loop in this fashion necessitates realizing that the loads and stores are independent and can be interchanged.

Summary of the Loop Unrolling and Scheduling

Throughout this chapter and Appendix H, we will look at a variety of hardware and software techniques that allow us to take advantage of instruction-level parallelism to fully utilize the potential of the functional units in a processor. The key to most of these techniques is to know when and how the ordering among instructions may be changed. In our example, we made many such changes, which to us, as human beings, were obviously allowable. In practice, this process must be performed in a methodical fashion either by a compiler or by hardware. To obtain the final unrolled code, we had to make the following decisions and transformations:

- Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.
- Use different registers to avoid unnecessary constraints that would be forced by using the same registers for different computations (e.g., name dependences).
- Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.
- Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent. This transformation requires analyzing the memory addresses and finding that they do not refer to the same address.
- Schedule the code, preserving any dependences needed to yield the same result as the original code.

The key requirement underlying all of these transformations is an understanding of how one instruction depends on another and how the instructions can be changed or reordered given the dependences.

Three different effects limit the gains from loop unrolling: (1) a decrease in the amount of overhead amortized with each unroll, (2) code size limitations, and (3) compiler limitations. Let's consider the question of loop overhead first. When we unrolled the loop four times, it generated sufficient parallelism among the instructions that the loop could be scheduled with no stall cycles. In fact, in 14 clock cycles, only 2 cycles were loop overhead: the `addi`, which maintains the index value, and the `bne`, which terminates the loop. If the loop is unrolled eight times, the overhead is reduced from 1/2 cycle per element to 1/4.

A second limit to unrolling is the resulting growth in code size. For larger loops, the code size growth may be a concern, particularly if it causes an increase in the instruction cache miss rate.

Another factor often more important than code size is the potential shortfall in registers that is created by aggressive unrolling and scheduling. This secondary effect that results from instruction scheduling in large code segments is called *register pressure*. It arises because scheduling code to increase ILP causes the number of live values to increase. After aggressive instruction scheduling, it may not be possible to allocate all the live values to registers. The transformed code, while theoretically faster, may lose some or all of its advantage because it leads to a shortage of registers. Without unrolling, aggressive scheduling is sufficiently limited by branches so that register pressure is rarely a problem. The combination of unrolling and aggressive scheduling can, however, cause this problem. The problem becomes especially challenging in multiple-issue processors that require the exposure of more independent instruction sequences whose execution can be overlapped. In general, the use of sophisticated high-level transformations, whose potential improvements are difficult to measure before detailed code generation, has led to significant increases in the complexity of modern compilers.

Loop unrolling is a simple but useful method for increasing the size of straight-line code fragments that can be scheduled effectively. This transformation is useful in a variety of processors, from simple pipelines like those we have examined so far to the multiple-issue superscalars and VLIWs explored later in this chapter.

3.3 Reducing Branch Costs With Advanced Branch Prediction

Because of the need to enforce control dependences through branch hazards and stalls, branches will hurt pipeline performance. Loop unrolling is one way to reduce the number of branch hazards; we can also reduce the performance losses of branches by predicting how they will behave. In Appendix C, we examine simple branch predictors that rely either on compile-time information or on the observed dynamic behavior of a single branch in isolation. As the number of instructions in flight has increased with deeper pipelines and more issues per clock, the importance of more accurate branch prediction has grown. In this section, we examine techniques for improving dynamic prediction accuracy. This section makes extensive use of the simple 2-bit predictor covered in Section C.2, and it is critical that the reader understand the operation of that predictor before proceeding.

Correlating Branch Predictors

The 2-bit predictor schemes in Appendix C use only the recent behavior of a single branch to predict the future behavior of that branch. It may be possible to improve the prediction accuracy if we also look at the recent behavior of *other* branches rather than just the branch we are trying to predict. Consider a small code fragment from the eqntott benchmark, a member of early SPEC benchmark suites that displayed particularly bad branch prediction behavior:

```
if (aa==2)
        aa=0;
if (bb==2)
        bb=0;
if (aa!=bb) {
```

Here is the RISC-V code that we would typically generate for this code fragment assuming that `aa` and `bb` are assigned to registers `x1` and `x2`:

```
     addi  x3,x1,-2
     bnez  x3,L1       //branch b1  (aa!=2)
     add   x1,x0,x0    //aa=0
L1:  addi  x3,x2,-2
     bnez  x3,L2       //branch b2  (bb!=2)
     add   x2,x0,x0    //bb=0
L2:  sub   x3,x1,x2    //x3=aa-bb
     beqz  x3,L3       //branch b3  (aa==bb)
```

Let's label these branches b1, b2, and b3. The key observation is that the behavior of branch b3 is correlated with the behavior of branches b1 and b2. Clearly, if neither branches b1 nor b2 are taken (i.e., if the conditions both evaluate to true and `aa` and `bb` are both assigned 0), then b3 will be taken, because `aa` and `bb` are clearly equal. A predictor that uses the behavior of only a single branch to predict the outcome of that branch can never capture this behavior.

Branch predictors that use the behavior of other branches to make a prediction are called *correlating predictors* or *two-level predictors*. Existing correlating predictors add information about the behavior of the most recent branches to decide how to predict a given branch. For example, a (1,2) predictor uses the behavior of the last branch to choose from among a pair of 2-bit branch predictors in predicting a particular branch. In the general case, an (*m*,*n*) predictor uses the behavior of the last *m* branches to choose from 2^m branch predictors, each of which is an *n*-bit predictor for a single branch. The attraction of this type of correlating branch predictor is that it can yield higher prediction rates than the 2-bit scheme and requires only a trivial amount of additional hardware.

The simplicity of the hardware comes from a simple observation: the global history of the most recent *m* branches can be recorded in an *m*-bit shift register, where each bit records whether the branch was taken or not taken. The branch-prediction buffer can then be indexed using a concatenation of the low-order bits from the branch address with the *m*-bit global history. For example, in a (2,2) buffer with 64 total entries, the 4 low-order address bits of the branch (word address) and the 2 global bits representing the behavior of the two most recently executed branches form a 6-bit index that can be used to index the 64 counters. By combining the local and global information by concatenation (or a simple hash function), we can index the predictor table with the result and get a prediction as fast as we could for the standard 2-bit predictor, as we will do very shortly.

How much better do the correlating branch predictors work when compared with the standard 2-bit scheme? To compare them fairly, we must compare predictors that use the same number of state bits. The number of bits in an (m,n) predictor is

$$2^m \times n \times \text{Number of prediction entries selected by the branch address}$$

A 2-bit predictor with no global history is simply a (0,2) predictor.

Example How many bits are in the (0,2) branch predictor with 4K entries? How many entries are in a (2,2) predictor with the same number of bits?

Answer The predictor with 4K entries has

$$2^0 \times 2 \times 4\text{K} = 8\text{K bits}$$

How many branch-selected entries are in a (2,2) predictor that has a total of 8K bits in the prediction buffer? We know that

$$2^2 \times 2 \times \text{Number of prediction entries selected by the branch} = 8\text{K}$$

Therefore the number of prediction entries selected by the branch = 1K.

Figure 3.3 compares the misprediction rates of the earlier (0,2) predictor with 4K entries and a (2,2) predictor with 1K entries. As you can see, this correlating predictor not only outperforms a simple 2-bit predictor with the same total number of state bits, but it also often outperforms a 2-bit predictor with an unlimited number of entries.

Perhaps the best-known example of a correlating predictor is McFarling's gshare predictor. In gshare the index is formed by combining the address of the branch and the most recent conditional branch outcomes using an exclusive-OR, which essentially acts as a hash of the branch address and the branch history. The hashed result is used to index a prediction array of 2-bit counters, as shown in Figure 3.4. The gshare predictor works remarkably well for a simple predictor, and is often used as the baseline for comparison with more sophisticated predictors. Predictors that combine local branch information and global branch history are also called *alloyed predictors or hybrid predictors*.

Tournament Predictors: Adaptively Combining Local and Global Predictors

The primary motivation for correlating branch predictors came from the observation that the standard 2-bit predictor, using only local information, failed on some important branches. Adding global history could help remedy this situation. *Tournament predictors* take this insight to the next level, by using multiple predictors, usually a global predictor and a local predictor, and choosing between them

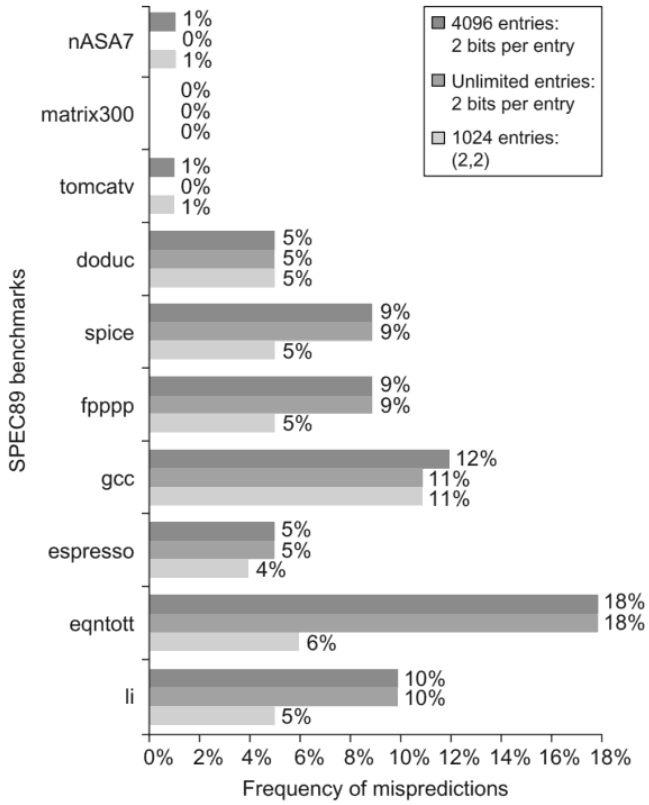


Figure 3.3 Comparison of 2-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating 2-bit predictor with unlimited entries and a 2-bit predictor with 2 bits of global history and a total of 1024 entries. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

with a selector, as shown in Figure 3.5. A *global predictor* uses the most recent branch history to index the predictor, while a *local predictor* uses the address of the branch as the index. Tournament predictors are another form of hybrid or alloyed predictors.

Tournament predictors can achieve better accuracy at medium sizes (8K–32K bits) and also effectively use very large numbers of prediction bits. Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor (local, global, or even some time-varying mix) was most effective in recent predictions. As in a simple 2-bit predictor, the saturating counter requires two mispredictions before changing the identity of the preferred predictor.

The advantage of a tournament predictor is its ability to select the right predictor for a particular branch, which is particularly crucial for the integer benchmarks.

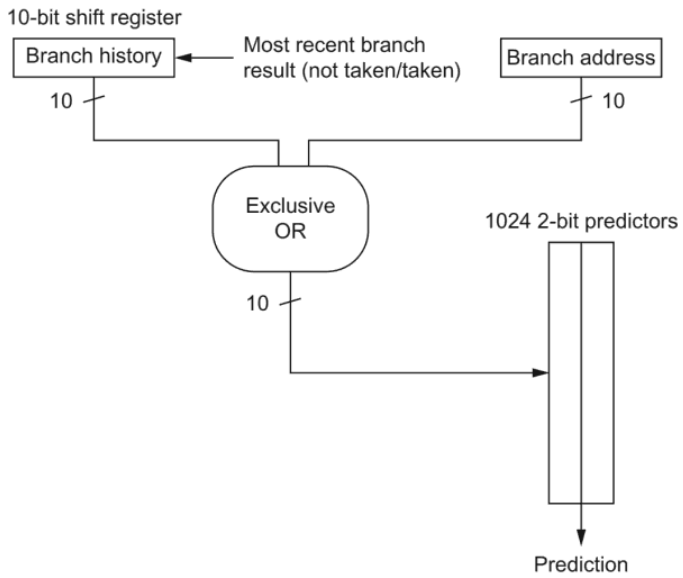


Figure 3.4 A gshare predictor with 1024 entries, each being a standard 2-bit predictor.

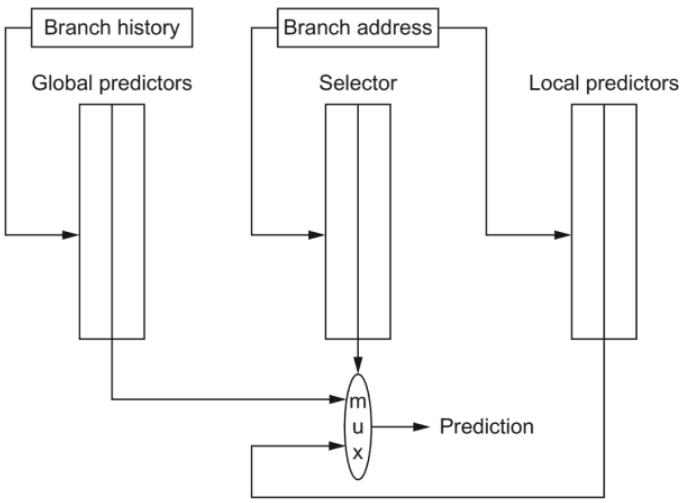


Figure 3.5 A tournament predictor using the branch address to index a set of 2-bit selection counters, which choose between a local and a global predictor. In this case, the index to the selector table is the current branch address. The two tables are also 2-bit predictors that are indexed by the global history and branch address, respectively. The selector acts like a 2-bit predictor, changing the preferred predictor for a branch address when two mispredicts occur in a row. The number of bits of the branch address used to index the selector table and the local predictor table is equal to the length of the global branch history used to index the global prediction table. Note that misprediction is a bit tricky because we need to change both the selector table and either the global or local predictor.

A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks. In addition to the Alpha processors that pioneered tournament predictors, several AMD processors have used tournament-style predictors.

Figure 3.6 looks at the performance of three different predictors (a local 2-bit predictor, a correlating predictor, and a tournament predictor) for different numbers of bits using SPEC89 as the benchmark. The local predictor reaches its limit first. The correlating predictor shows a significant improvement, and the tournament predictor generates a slightly better performance. For more recent versions of the SPEC, the results would be similar, but the asymptotic behavior would not be reached until slightly larger predictor sizes.

The local predictor consists of a two-level predictor. The top level is a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. That is, if the branch is taken 10 or more times in a row, the entry in the local history table will be all 1s. If the branch is alternately taken and untaken, the history entry consists of alternating 0s and 1s. This 10-bit history allows patterns of up to 10 branches to be discovered and predicted. The selected entry from the local history table is used to index a table of 1K entries consisting of 3-bit saturating counters, which provide the local prediction. This combination, which uses a total of 29K bits, leads to high accuracy in

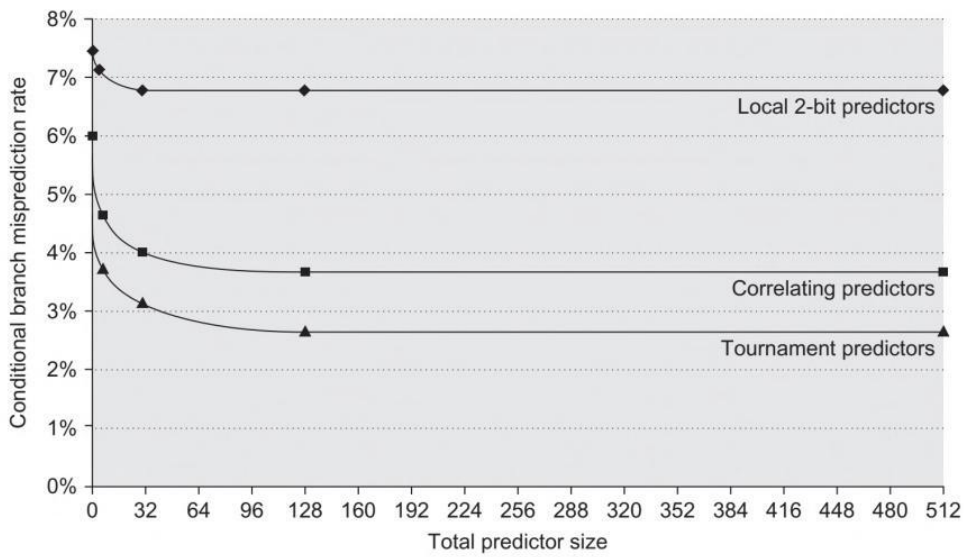


Figure 3.6 The misprediction rate for three different predictors on SPEC89 versus the size of the predictor in kilobits. The predictors are a local 2-bit predictor, a correlating predictor that is optimally structured in its use of global and local information at each point in the graph, and a tournament predictor. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks show similar behavior, perhaps converging to the asymptotic limit at slightly larger predictor sizes.